2 of 3

THE SYNTHESIS OF REDUNDANT MULTIPLE — LINE NETWORKS

Second Annual Report
Contract Nonr 3842 (00)
May 1, 1964

by
W. L. Glasgrove
R. A. Jensen
W. C. Mann
G. M. Schechter

WESTINGHOUSE DEFENSE AND SPACE CENTER
SURFACE DIVISION
ADVANCED DEVELOPMENT ENGINEERING

P. O. Box 1897 Baltimore 3, Maryland

Second Annual Report

Contract Nonr 3842 (00)

For the Period May 1, 1963 to May 1, 1964

THE SYNTHESIS OF REDUNDANT MULTIPLE-LINE NETWORKS

May 1, 1964

Prepared for the

Office of Naval Research

by

Westinghouse Electric Corporation

Surface Division

P. O. Box 1897

Baltimore 3, Maryland

Approved by:

S. B. Lomax, Director

ABSTRACT

This report is concerned with multiple-line redundancy, a means of increasing the reliability of electronic systems. It describes in detail a procedure for synthesizing redundant systems in an optimum manner. The procedure balances the reliability advantage of redundancy against the cost, weight and power penalties it introduces. It is also applicable to other problems which require the optimization of a large number of binary decisions. The report describes a program which has been written to implement the procedure on a digital computer. The program has been run successfully on example networks and is now ready for specific application.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF APPENDICES

LIST OF ILLUSTRATIONS

I. INTRODUCTION

Every electronic component is subject to failure, although modern technology has reduced the rate of this failure to extremely low levels. As modern warfare and data processing require machines which perform more and more sophisticated tasks, the electronics industry responds with extremely complex equipment requiring prodigious numbers of parts. Since a nonredundant machine requires the correct functioning of all its components, the individual small probabilities of failure accumulate to yield a very significant probability of failure for the equipment, causing an average time between failures of only a matter of hours. For the repairable machine, this means down time while repair is effected. For the nonrepairable machine, such as might be found in an unmanned orbital satellite or a ballistic missile guidance system, it means failure of a mission.

Often the down time associated with repair or the failure of a mission cannot be allowed or, at least, is extremely expensive. To overcome failure of modern electronic equipment, the use of redundancy has been proposed. In general, the term redundancy refers to extra equipment incorporated into the system which is above and beyond the minimum required to implement the task. This additional equipment is merged into the system such that failures are masked or overcome and the system operation is maintained even though a number of circuit failures have occurred.

This study has dealt with multiple-line redundancy, which is described in detail in Section II of this report. Westinghouse has studied several schemes for incorporating circuit redundancy into digital machines, and this has been found to be the most effective.

Multiple-line redundancy operates, in parallel, a number of replicas of each circuit in the nonredundant network. The number of replicas of a circuit is its order of redundancy. Groups of circuits called restorers, whose sole purpose is the correction of errors which arise due to circuit failures, are placed at various points in the network. The redundancy of circuits plus the restorers provide a network with an ability to withstand a number of circuit failures without impairment of its operation.

Past studies by Westinghouse, and a number of other investigations, have shown that multiple-line redundancy is indeed a valid approach to increasing the reliability of electronic equipment. This study is devoted to establishing procedures with which the designer can determine the best way to incorporate redundancy. It seeks to answer the question, "Given the nonredundant network, the reliability and cost of all its parts and the reliability and cost of restorers, what is the optimum way to assign redundancy to the circuits and what is the optimum way to place restorers in the network?" This is the problem of synthesis.

The first step toward a procedure to perform synthesis is the proposal of factors which are to be considered in determining whether one network design is better than another. This study has combined the factors of cost of the circuits in the network, reliability, weight, and power into a single criterion for optimization which is called the True Cost. The network which has the minimum True Cost is the optimum network.

The most obvious approach to finding the optimum network design is to try all the alternatives, measuring their True Costs, and picking out the most inexpensive design. Unfortunately, the number of alternatives one has to consider increases so rapidly with the size of the network that this exhaustive search approach is eminently impracticable for all but the smallest networks. What is desired from this study is a synthesis procedure which is deterministic, in that it gives the network that minimizes the True Cost, and which can be performed in a reasonable amount of time with the aid of a computer.

This report describes a procedure called the "Isolating Array Synthesis Procedure" which uses a characteristic of multiple-line networks to considerably reduce the number of calculations from the amount required for exhaustive search. At this point in the study, the same number of replicas must be provided for all circuits. The procedure is deterministic, but there are approximations inherent in its operation, and the result may not be the design which minimizes the True Cost. The approximations are small, however, and for most problems, the True Cost of the result of the Procedure will be very little greater than the minimum True Cost. The approximations are justified by the considerable savings in effort that are available through the use of the procedure. The result of this study is a significant contribution and is the first means proposed for finding the optimum arrangement of restorers short of an exhaustive search.

The basic concepts of the Isolating Array Synthesis Procedure were introduced in the First Annual Report (1) along with a new reliability analysis technique for multiple line networks. Most of the work during the last year was concerned with developing the synthesis procedure in sufficient detail so that it could be implemented on a digital computer. In the course of these studies a number of modifications and simplifications were incorporated into the procedure that were not present in the version described in the First Annual Report, hence, this report supercedes the first one. This report is intended to be a complete, self-contained description of the synthesis procedure, therefore it includes most of the information previously reported except for the reliability analysis technique.

II. DEFINITIONS

A. MULTIPLE-LINE REDUNDANCY

The type of network under study is shown in figure 2-1. With the help of this figure, several terms used in this report may be defined. The rectangles in the figure are non redundant digital circuits operating on binary information. They are numbered for identification. The dots in the rectangles indicate the order of redundancy in this case, one. Although the complexity of the circuits is not strictly limited in this discussion, the most exact representation of the network results if the circuits are as simple as possible, performing basic logical operations such as AND, OR, NOR, NOT or the sequential functions of flip-flops or other memory devices. The line segments represent connections between circuits. A line segment into a rectangle is an input to the circuit and a line segment out of a rectangle is an output.

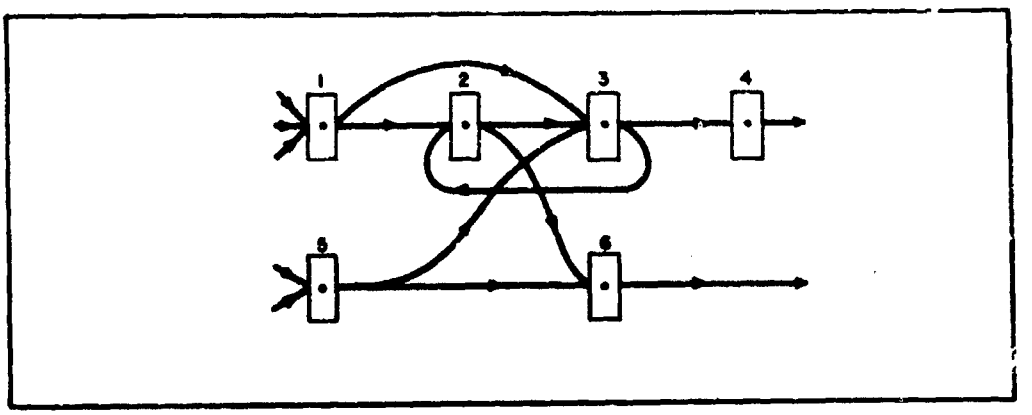


Figure 2-1. Example of a Nonredundant Network

The type of redundancy utilized in this report is one studied extensively by Westinghouse and found to be one of the most efficient types of circuit redundancy. It has been called multiple-line redundancy and is illustrated by figure 2-2.

In general, multiple-line redundancy is applied by replacing the single circuit of the nonredundant network by m identical circuits operating in parallel. The symbol m is called the order of redundancy. For the example m is 3. The group of circuits is now called a function.

A particular circuit in a redundant function is identified by its position. The lower case subscript on the numeral identifying each circuit is the position of that circuit.

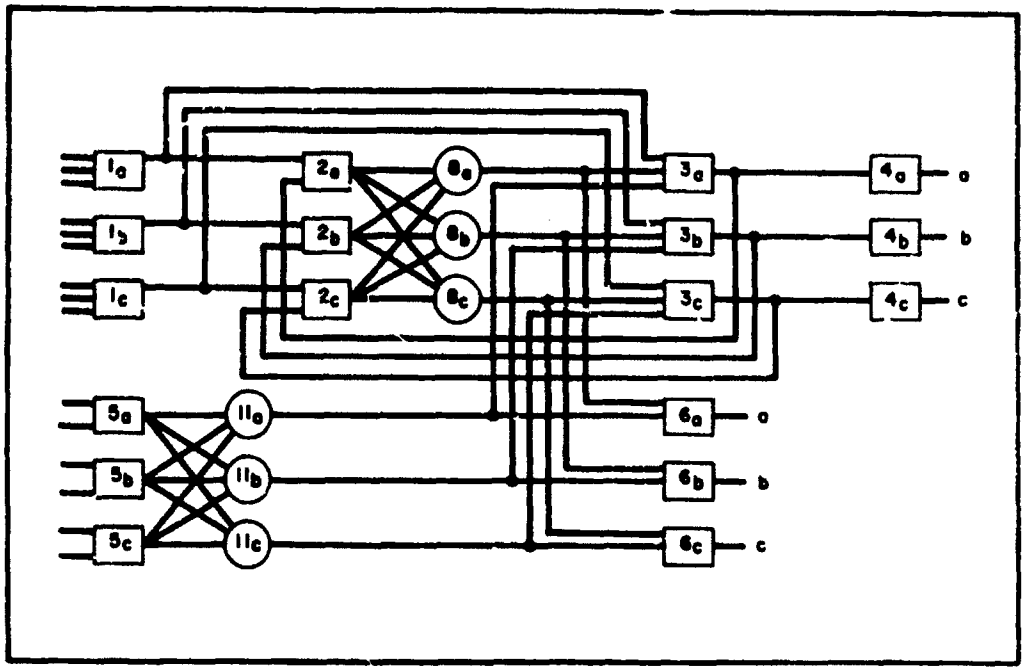


Figure 2-2. Order 3-Multiple-Line Redundant Network

Circuits or networks that are operating correctly or lines which carry correct information are said to be successful. The opposite states of incorrect operation or information are called failed.

The reliability improvement expected with the use of redundant circuits depends on the ability of the network to experience circuit failures without degradation of the network operation. The use of restorers within the network provides this characteristic. The sets of three circles in the figure represent restorers.

The restorer consists of m restoring circuits. If a restoring circuit is operating correctly, it has the ability to derive the correct output if k of its m inputs are correct. The restoring circuits for the example are majority gates with $k = 2$ and $m = 3$. Working restoring circuits filter out errors on their inputs. The only reason for an erroneous output line of a restorer is the failure of a restoring circuit or the incidence of $m - k + 1$ or more errors on the inputs to the restorer. In the event of the latter condition, all the restorer outputs become erroneous since the restoring circuits are identical. Restoring circuit reliability is assumed to be independent of whether its inputs include failures. If the restoring circuit has at least k correct inputs, the probability that its output is correct or failed depends only on the reliability of the restoring circuit.

When the functions at the restorer's input and output are the same order of redundancy, it has m inputs and m outputs. Its inputs are the outputs of one function, and its outputs provide the inputs to one or more functions.

Restorers may operate on the output lines of a function, as described in this section or on the input lines to a function as described in references (2), (3), and (4). Studies at Westinghouse, Hycon Eastern[3], and at IBM[4] indicate the former arrangement is most effective. Although it is difficult to prove in general that output voting is always superior to input voting, this has been the case for every specific example chosen. This report, therefore, will assume that all networks are constructed utilizing output voting.

A function which has a restorer on its output is called a restored function. Errors on the output of a restored function can be corrected if at least k of the m output lines are successful. This report assumes that a means is available to correct errors on network outputs, such that only k of the m output lines need be successful for the network to be operating successfully. The functions providing network outputs, then, are also classed as restored functions.

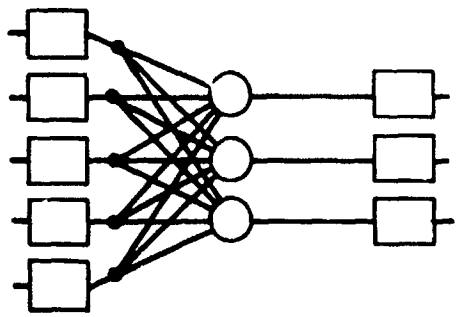
With s functions in the network, numbers from 1 to s identify the functions. There is a possible location of a restorer at the output of each function; it may or may not contain a restorer. A restorer after function y is identified by the number $s + y$. Thus, in this example in which $s = 6$, restorer 8 operates on the output lines of function 2 and restorer 11 operates on the output lines of function 5.

Two assumptions on the effects of circuit failures are made for this report. First, when a circuit (in a function or a restorer) fails, its output is always in error. Secondly, when a circuit in a function has an input which is in the failed condition, the output of that circuit is failed. For instance, if circuit 1_a is failed, its own output and the outputs of the circuits 2_a, 3_a, and 4_a are in error.

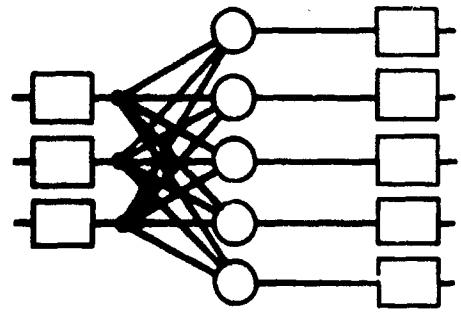
Functions are interconnected in the manner shown in figure 2-2. In order that failure of one circuit in any function does not disable the inputs to two circuits in another function, only circuits in the same position should be interconnected.

Different functions may have different orders of redundancy, but the order of redundancy may change only after passing through a restorer. Figure 2-3 shows the connection between functions with different orders or redundancy.

Figure 2-4 gives a more condensed view of the network of figure 2-2. This type of diagram will be used throughout the remainder of the report. The circles represent restorers and the three dots in each symbol indicate that this network is order three redundancy



a) GOING FROM ORDER 5 REDUNDANCY TO ORDER 3.



b) GOING FROM 3 TO 5

Figure 2-3. Restoration Between Different Orders of Redundancy

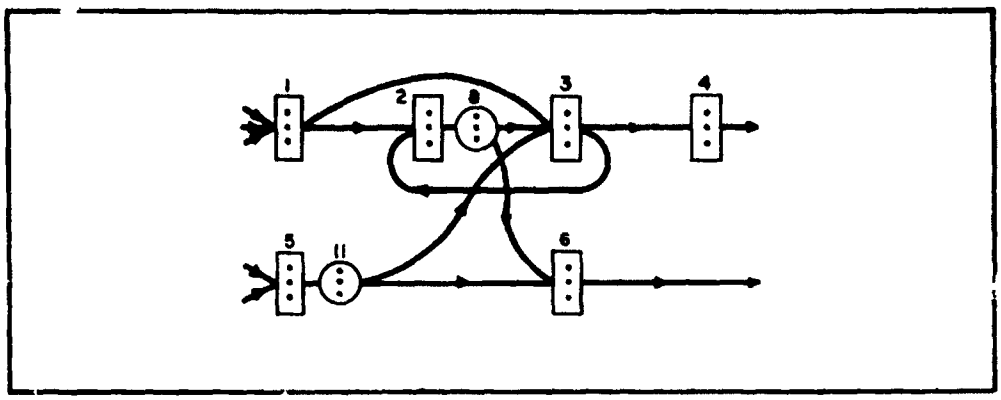


Figure 2-4. Flow Graph Representing a Redundant Network

throughout. This diagram is used to introduce some terms peculiar to this report which find considerable use later.

A source of a redundant function y is a function or restorer x so connected to y that a path can be traced from the output of x to the output of y along the directed line segments of the diagram. As an example of sources, consider function 6 in figure 2-4. The sources of function 6 are restorers 8 and 11 and functions 1, 2, 3 and 5. There is no path between function 4 and function 6.

A function may have primary, secondary or higher order sources depending on the number of line segments traversed on the diagram when going from the source to the function.

A term which describes the same relationship as source but which eases considerably future descriptions is sink. If function x is said to be a source of y, function y can be called a sink of x. Of course functions or restorers may have primary, secondary or higher order sinks.

B. ERROR-LINKED AND ISOLATED FUNCTIONS

1. Error-Linked Functions

In a multiple-line redundant network two functions or restorers are related to each other by the effect of failures in both of them on the operation of the network. Two terms are defined here which describe opposite effects.

Two functions, two restorers or a function and restorer are said to be error-linked if failures in one can combine with failures in another to cause network failure, even though the failures in either alone are insufficient to cause network failure.

There are two types of error-linking that may occur. Consider the order three network in figure 2-2. Of course for this order of redundancy two failures are sufficient to cause the network to fail. The flow graph for this example is found in figure 2-4. A signal flow path is a path along the directed line segments in the direction of the arrow heads. A restorer interrupts a signal flow path. The first type of error-linking occurs between two functions which are on the same signal flow path of the flow graph. For instance functions 1 and 3 are error linked because failure of circuit 1a and circuit 3b will cause the output of function 3 to be in error on a majority of lines. Error-linking arises in this manner only if one function is upstream (against the signal flow) from the other; hence it is referred to as error-linking from the upstream effect. Function 1 is upstream from function 3. In like manner restorers 8 and 11 are also error-linked to function 3.

The second type of error-linking occurs when failures in two functions (or restorers, not on the same signal flow path combine to cause network failures. In the example, function 1 and restorer 11 are error-linked in this way. For this type of error-linking to occur both functions must feed a third function, or, in other words, there is a common function downstream (with the direction of signal flow) from both error linked functions. Error-linking in this manner is referred to as error linking from the downstream effect.

There is no direction implied when function a is said to be error linked to function b. The statement only indicates that circuit failures in the two functions can combine to cause network failure. The two statements, a is error-linked to b, and b is error-linked to a are equivalent.

2. Isolated Functions

When two functions are not error-linked they are isolated. Two functions or restorers are isolated if a circuit failure in one does not affect the outputs of any of the same functions as a circuit failure in the other. In figure 2-4, function 5 is isolated from every other function and restorer in the network.

Functions may be isolated from each other in two ways. First, in a network with more than one output, two functions may be isolated by the form of the network. In figure 2-5, functions 3 and 4 form the outputs of a redundant network. No error in a circuit in function 3 can combine with an error in function 4 to cause failure of the network.

Secondly, restorers isolate functions. For instance functions 1 and 2 in the shift register of figure 2-6 are isolated from functions 3, 4, 5 and 6 by the restorer in location 2.

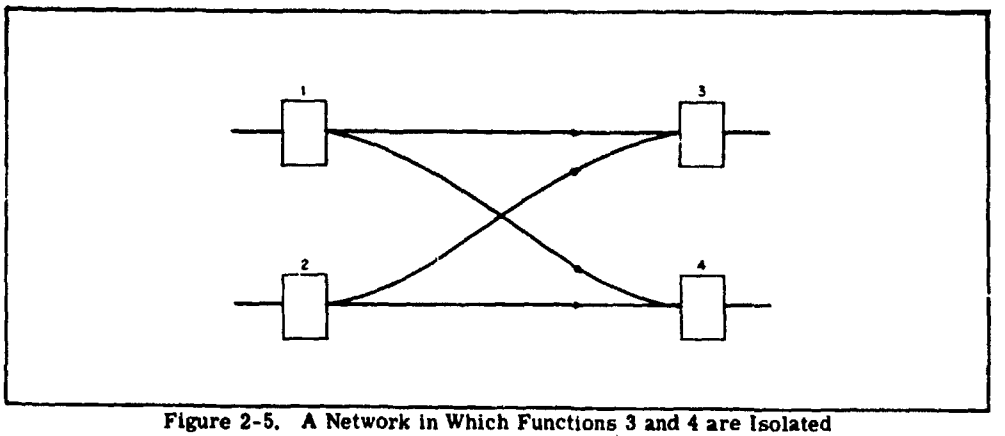


Figure 2-5. A Network in Which Functions 3 and 4 are Isolated

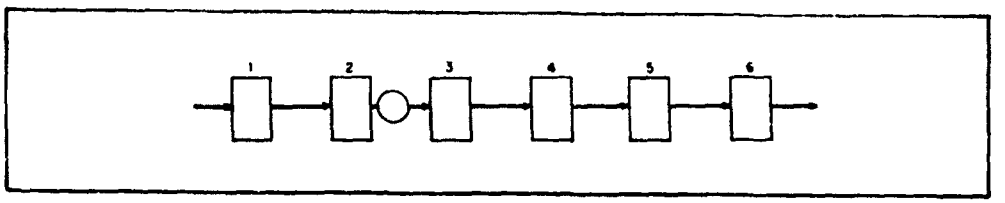


Figure 2-6. A Network in Which Functions 1 and 2 are Isolated from Functions 3, 4, 5 and 6

As long as there are k or more correct inputs to a restorer, errors are not transmitted from its inputs to its outputs. For instance, a single erroneous input to the three input majority gate of the order three restorer has no effect on the output of the gate. Thus, since there is a restorer at location 2, single errors in 1 and 3 cannot combine to cause network failure.

The concept of isolation is important to the synthesis procedure because it describes the condition for independence of reliability between two isolated regions. For instance, in figure 2-6 functions 1 and 2 are isolated from functions 3, 4, 5 and 6. The first two functions form an isolated region and the restorer and the latter 4 functions form another. Since failures in different isolated regions cannot combine to cause network failure, the reliabilities of the regions are independent. Hence if R_1 is the reliability of the first region and R_2 the reliability of the second, the probability that neither region is failed is R_1R_2.

3. Isolated and Error-Linked Sources and Sinks

With isolation and error-linking defined, the sources and sinks of a function in a redundant multiple-line network fall into two classes. The sources and sinks are either

isolated from the function or error-linked to the function. This distinction finds considerable application in the synthesis procedure.

C. THE ARRANGEMENT OF RESTORERS AND FUNCTIONS

This section introduces some terms and notation which are used to specify arrangements of restorers or groups of functions in a network. They will find considerable application as this report progresses.

1. State of a Location

The state of a location indicates whether a restorer is present or not present in that location. If a restorer is present in location i, location i is said to be filled and its state is a binary 1. If no restorer is present, location i is said to be empty and its state is a binary 0.

In general, a binary variable x_i represents the state of the ith location.

2. Array

During the discussion to follow it will often be necessary to refer to the states of a set of locations (not necessarily all locations in the network). The general term referring to the states of the locations in such a set is array. An array is defined as a set of filled and empty locations. The locations and their states completely specify an array.

The term array will also be used to describe a set of locations some of which have variable states. Of course the variables associated with such locations will be binary.

The array which specifies the states of all the locations in the network as filled or empty takes the special designation network array. Each network array represents a possible design of the redundant network.

3. Array Vectors

Vector notation is used to specify the states of the locations in an array. The binary variable x_i defines the state of the ith location, and a vector, using as coordinates the variables representing the s locations in the network, designates a network array.

$$(x_1, x_2, x_3, \ldots, x_s)$$

Each set of values assumed by the binary variables which are the coordinates of this vector represents a different network array or network design. With s coordinates, there are 2^s different vectors described by the general vector, hence this is the total number of restorer arrangements applicable to the network.

Arrays which do not include all the network's locations also are identified with the vector notation. The coordinates representing the locations not in the array are not identifiec in the vector by a 0 or 1 but remain as an x. For instance, a network with five locations, numbered 1 through 5, has an array in which locations 1 and 2 are filled, 3 and 4 are empty, and location 5 is not in the array. The vector representation of this array is:

(1, 1, 0, 0, x).

No subscript on the x in this vector is necessary. The position of the coordinate in the vector identifies the corresponding location.

An array which excludes one or more locations really is representing a number of network arrays. The specification of the states of less than the total number of locations makes the unspecified locations arbitrary and allows them to assume any value. The vector (1, 1, 0, 0, x) represents two vectors, (1, 1, 0, 0, 0) and (1, 1, 0, 0, 1). In general, if an array does not specify z locations, the number of network arrays it identifies is 2^z.

4. Region

Region is a general term referring to a specified set of functions and restorers. Generally, a region is defined by some characteristic such as "all functions and only those functions that are error-linked to function A are members of the region."

5. Isolating Arrays and Isolated Regions

Two very important concepts which find considerable application are isolating arrays and isolated regions. These should be clearly understood before the detailed procedure is described.

An isolated region or an isolating array is defined for a given group of functions, so these functions must be specified along with the region or array. Suppose the group of functions is the set A. An isolated region of the functions in set A consists of a set of functions and restorers B, which includes the functions in A, such that all functions not in B are isolated from the functions in A. For instance, in figure 2-7 are illustrated three isolated regions of functions 1 and 2 (set A). The functions and restorers that make up each region (set B) are also shown in the figure.

Note in the last example of figure 2-7 that not all functions in the region are error linked to functions 1 or 2. In general it is not required that all functions in the region be error linked to the functions in the set A.

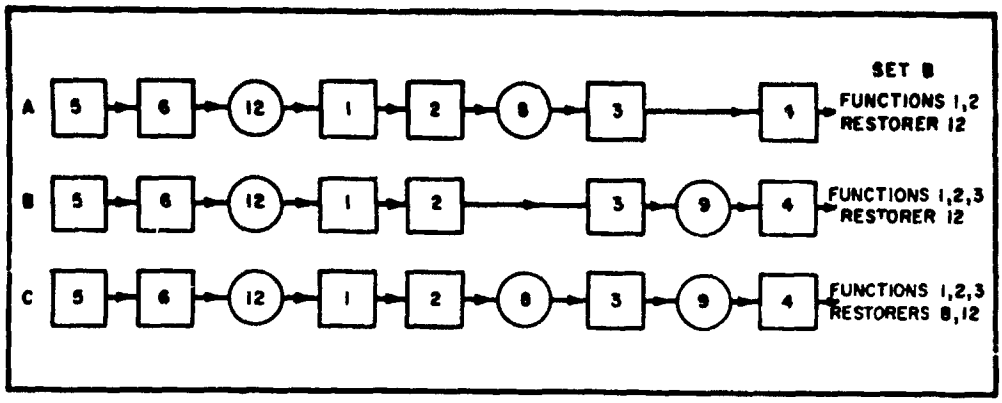


Figure 2-7. Three Examples of Isolated Regions of Functions 1 and 2

For each isolated region there is one and only one isolating array which, together with the identity of the given group of functions, completely defines the isolated region. The array specifies as 1 the locations at the boundaries of the region in which restorers are required to isolate functions outside the region from the given group of functions. Also specified as either 1 or 0, are all the locations of the functions within the isolated region. The states of the locations not necessary to define the region are specified as x's in the isolating array. Only locations required to identify the members of an isolated region are specified in the isolating array. The isolating arrays of the regions shown in figure 2-7 are:

A. (01xxx1)
B. (001xx1)
C. (011xx1).

III. THE SYNTHESIS OF REDUNDANT NETWORKS

A. GENERAL

The goal of this study is the development of a synthesis procedure by which the designer of a redundant multiple-line system can determine in some optimum manner the orders of redundancy of the functions and the placement of restorers in the system. This will be an important accomplishment because it can be shown that redundancy in the wrong places can be almost useless and that an improperly placed restorer is sometimes worse than no restorer at all. The most beneficial synthesis procedure would be one which allowed full flexibility in the order of redundancy of the functions and placement of restorers, which was deterministic, in that it resulted in one redundant network which was optimum according to some useful criterion, and which was easily performed in a reasonable amount of time with the help of a computer.

The goal, as stated in the last paragraph, is an extremely difficult one to attain and it has been compromised at this point only to bring the problem into a still complex but solvable form. The restrictions made for this synthesis procedure are not expected to be permanent. Future studies will attempt to remove them.

The primary restriction on the network is that all the functions must be in the same order of redundancy. This reduces the synthesis problem to finding the proper placement of restorers. This is still a significant problem since in an s function network there are 2^s possible restorer arrangements that can be applied to the network.

The procedures are derived with computer implementation in mind. In most cases, the number of calculations required for the synthesis of large networks will be small relative to the number required for an exhaustive search procedure. The number will still be great enough, however, to make prohibitive the performance of synthesis by hand for all but the smallest networks. A computer has been programmed to rapidly determine the optimum arrangement of restorers in the network.

The following paragraphs of this section will discuss the optimization criterion and the general principles of the synthesis procedure.

B. OPTIMIZATION CRITERION

Each of the 2^s possible network array vectors represents a different design of the redundant system. To choose one of these as a best design, one must have some criterion with which the many alternative networks may be compared.

Of course, reliability is the first criterion since the redundancy has been introduced to increase this vital parameter. The cost of the circuitry required to implement a particular redundant design may also be of importance. For many applications, the weight and power requirements of alternatives will very probably enter into consideration. The factors of reliability, cost of implementation, weight and power requirements may all enter into the decision determining the best or optimum design. Other factors may also be significant and should be considered in the same manner as the assumed factors in the criterion presented below.

To optimize the network with respect to any one of these factors is to suboptimize with respect to all the others, so this study lumps all of them into a single cost expression. The goal of the synthesis procedure is to find the network which minimizes this cost.

The reliability of the network enters into the cost expression as the cost of failure of the network. A failure will always be costly. If this were not so, there would be no point in incorporating redundancy.

Where the application of the network is a control function in a satellite or rocket or where human life is concerned, this cost of failure is exceedingly high and probably overrides the other factors. On the other hand, if the network is to be utilized for a ground based computing system, this cost although high, may be low enough so that the other factors enter into consideration. If K is said to be the cost of failure of the network and R is its reliability, the expected cost due to failure is:

$$(1-R)K. \tag{1}$$

The cost of implementing a particular redundant design is assumed to be linearly dependent on the order of redundancy of the functions and the number of restorers in the network. Letting m_i be the order of redundancy C_{fi} be the cost of a circuit in the i function or restorer, the implementation of a redundant network requires the expenditure of

$$\sum^{\text{all } i} m_i C_{fi}. \tag{2}$$

The "all i" statement over the summation sign indicates that the sum is over all the functions and restorers in the network.

The cost equation reflects weight and power penalties by introducing per unit costs for these parameters. The weight added by one circuit of the type used in function i is W_i and the cost per pound of weight in function i is C_{Wi}. The cost associated with the weight of the network can be written:

$$\sum^{\text{all } i} m_i W_i C_{Wi}. \tag{3}$$

Describing the costs associated with power in the same manner and summing the terms to obtain the total cost due to power and weight one obtains:

$$\sum^{\text{all } l} m_l \left[W_l C_{Wl} + P_l C_{Pl} \right] . \qquad (4)$$

A number of other factors which the designer might like to consider in the optimization can be included in a manner similar to weight and power.

Terms (1), (2) and (4) summed to a single cost expression is called the True Cost.

$$\text{True Cost} = \sum^{\text{all } l} m_l (C_{Il} + W_l C_{Wl} + P_l C_{Pl}) + (1-R)K. \qquad (5)$$

The term furthest to the right in equation (5), which is concerned with reliability, is called the expected cost of failure of the network. The sum of the remainder of the terms dealing with the costs of implementation, weight, power and any other linear nonreliability factors is called the functional cost of the network. The network array for which the True Cost is least is the True Optimum.

This report uses an approximation to equation (5) as the criterion for optimization.

C. THE COEFFICIENTS OF THE TRUE COST POLYNOMIAL

It is important that one consider the ramifications of choosing the true cost as a basis for synthesis. Before accepting the use of this parameter the meaning of constant and nonconstant coefficients should be made clear. Some answer must be given the questions "Can synthesis be performed in the face of nonconstant coefficients?" and "If constant coefficients are present, how can they be determined?" This section attempts to clarify these questions.

1. Constant Coefficients

The synthesis procedure, as it is described in this report assumes that the coefficients of the true cost polynomial are nonvariant as the variables of the synthesis (weight, power, cost, reliability) take on different values. Thus the cost of adding a restorer to location i does not depend on the total weight, power, cost or reliability of the network because the coefficients C_{Wl}, W_l, C_{Pl}, P_l, C_{Il} and K do not depend on these factors.

If one considers a system as made up of a number of subsystems, each of which is isolated from each other in the manner of section II. B. 2, the reliability of each subsystem can be determined independently of all others. With constant coefficients each subsystem can be optimized independently of all others. The combination of the optimum subsystems will be the optimum system.

Although the values of the coefficients cannot change with the variables of the design, the true cost polynomial does allow differences between the coefficients assigned to different functions. Thus C_{Ii}, C_{W1}, W_1, C_{P1} and P_1 may be different from C_{Ii}, C_{W2}, W_2, C_{P2}, P_2 respectively. Certainly one would expect variations of C_{Ii}, W_i and P_i from function to function, but variations of C_{W1} and C_{P1} are liable to be more infrequent. One case where the latter coefficients might be different for different parts of the system occurs when one subsystem is located on the ground and another in a space craft. Certainly differences will be present between the cost/lb. and cost/watt of these subsystems.

Some variation of the cost of failure, K, is allowed within the structure of the True Cost polynomial. Different subsystems may have different costs of failure if their reliabilities are always determined independently from each other during the course of design. For instance, taking again the example with a subsystem on the ground and one in a space craft, the reliabilities of these two subsystems can very probably be determined independently hence they can be given different costs of failure. This is a very useful facility. This situation will be handled in the True Cost calculation by including two expected cost of failure terms, one for subsystem 1 and one for subsystem 2 as shown below:

$$K_1(1-R_1) + K_2(1-R_2).$$

There is some approximation in this formulation. It is the same sort of approximation as described in Section IIIF.

2. Non-Constant Coefficients

Under the influence of non-constant coefficients the amount added to the true cost when restorer $\underline{i}$ is included in the network will depend on the value of weight, power cost and/or reliability of the remainder of the system. One example of such a situation is the condition in which any arrangement of restorers is allowed as long as the weight, power, cost and unreliability remain below upper bounds. Thus a restorer costs nothing if it results in an acceptable system, and it is not allowed, or has an infinite cost, if its addition results in an unacceptable system.

In this day of highly complex systems, it is common practice to divide systems into subsystems with a team of engineers responsible for the development of each subsystem. Frequently the design of each subsystem must meet restrictions on weight, power, cost and reliability allocated by the management responsible for the system as a whole. Whether or not this is a wise practice, it will result in non-constant coefficients of the true cost equation. The designer faced with such restrictions cannot assume constant coefficients.

The synthesis procedure described herein cannot be applied a single time to determine the optimum network when coefficients of the true cost equations are not constant. It can be applied, however, a number of times to provide the design which is optimum within restrictions. The use of the procedure under these conditions is described in the next section.

To design with non-constant coefficients, one must consider the system for which they were determined. Assume a system A exists which can be subdivided into two independent subsystems A' and A". Say for the system A the total cost of weight $\sum_{\text{all } i} m_i C_{Wi} W_i$ varies with weight in a nonlinear manner as shown in figure 3-1. This is equivalent to saying that there is a non-constant weight coefficient.

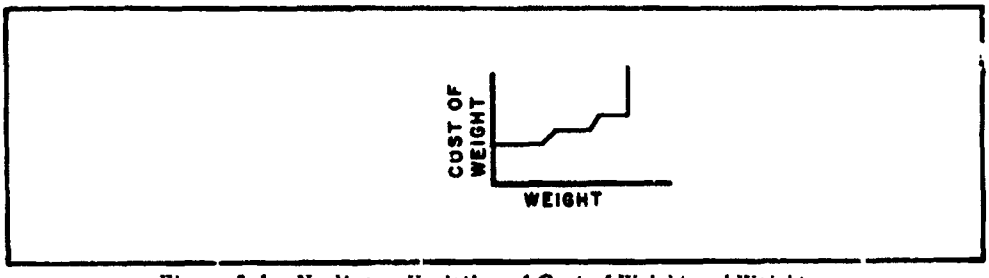


Figure 3-1. Nonlinear Variation of Cost of Weight and Weight

Now the variation of the cost with weight has been defined only for the total system, hence the variation of cost with weight for either subsystem is undefined. It is therefore impossible to optimize A' and A" independently using the information of figure 3-1. Thus the conclusion can be drawn that non-constant coefficients derived for a system can only be used for the optimization of that system in toto. They cannot be used for the optimization of any subsystem independently of the rest of the system.

3. Designing in the Face of Nonconstant Coefficients

There are two end results which one might hope to achieve when designing in the face of non-constant coefficients. The first of these will be to find the design which minimizes the true cost even though the true cost must be determined from relationships such as that of figure 3-1. The second will be to minimize one parameter (weight, power, probability of failure or cost) while keeping the other parameters within upper bound restrictions.

The solutions for either of these end results have a common characteristic: Any one of the parameters of the solution has the lowest value possible for the choice of the other parameters. For instance an optimum solution for a network which has cost, weight and probability of failure as parameters might have C_{opt} W_{opt} and $(1-P)_{opt}$ as the values of its parameters. It can be said that this optimum design has the lowest value of probability of

failure for designs with the cost, C_{opt}, and weight, W_{opt}. In like manner it has the smallest weight of all designs which have the parameter values $(1-P)_{opt}$ and C_{opt}, and it has the least cost of all designs which have the parameter values $(1-P)_{opt}$ and W_{opt}.

The solutions of the isolating array synthesis procedure with constant coefficients also have this characteristic. Thus the optimum design of a system which has non-constant coefficients must also be an optimum design of the same system when some set of constant coefficients are assigned. To synthesize the optimum network with non-constant coefficients one must find the proper set of constant coefficients.

This will be done by a number of applications of the synthesis procedure with different sets of constant coefficients. The process of finding the optimum will be a search procedure in which judicious choice of changes in the constant coefficients should lead to a more rapid convergence toward the optimum. The search procedure will receive more study in the future.

To see how multiple applications of the synthesis procedure can be used to construct tradeoff curves between the parameters of the system which will ultimately be used to design a network with nonconstant coefficients, define an optimal network as one which has one parameter minimized while all others are held constant. As described above the products of the synthesis procedure with constant coefficients are these optimal networks. By varying the coefficients of the true cost equation one will arrive at a number of different optimal networks. The values of the parameters of these optimal networks are the tradeoffs to be used in the design of a redundant network.

Thus the synthesis procedure has the ability to determine curves showing the tradeoffs between the parameters of optimal networks. The set of curves describes a multidimensional surface on which the parameters of all optimal networks must fall. The number of dimensions of the surface is the number of parameters of the network. For instance with the parameters weight, power, and failure probability, continued application of the synthesis procedure will yield a three dimensional surface on which all optimal networks must lie. A plane passed through this surface perpendicular to the power axis gives the two dimensional tradeoff curve for weight vs. failure probability for a single value of power. If one were to choose a value of weight which he is willing to expend, he determines from this curve the minimum failure probability that can be obtained for the given expenditures of weight and power. A typical set of tradeoff curves are shown in figure 3-2.

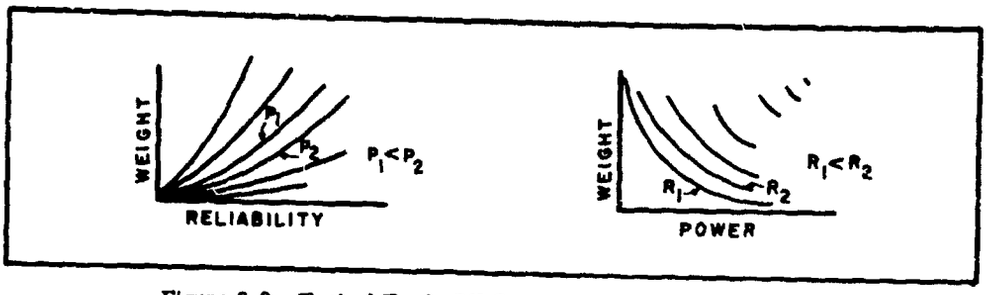


Figure 3-2. Typical Trade Off Curves for a Redundant System

To illustrate the optimization in the face of non-constant coefficients assume the system is subject to maximum weight and power constraints such that cost of weight and power is zero if these parameters lie below the limit and infinite if they lie above. The optimum network is found by drawing these limits on the weight and power curves as in figure 3-3.

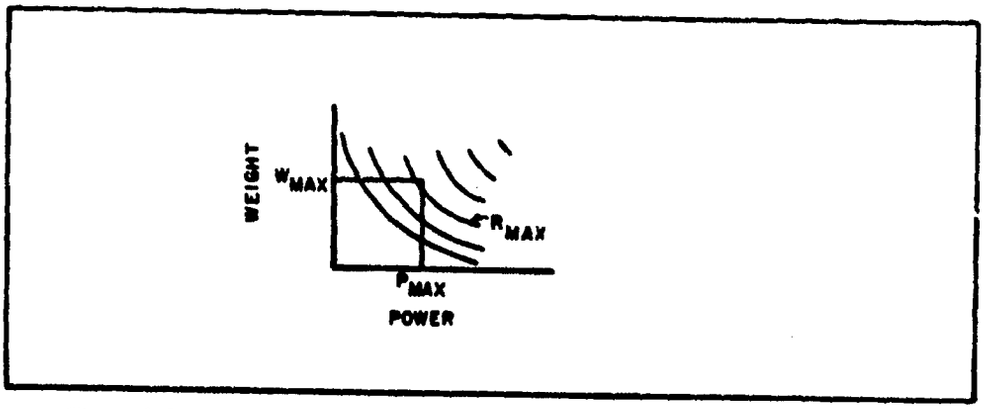


Figure 3-3. Finding the Optimum Network Under Weight and Power Constraints

The maximum reliability that one can attain under these constraints is the reliability curve furthermost from the origin that passes within the limits. This is indicated by R_{MAX} in the figure.

Note that the construction of the tradeoff curves does not require the actual values of cost per pound of weight, per watt of power, etc. Only when a final design must be chosen do the relationships between the parameters and cost come into play. The information contained in the tradeoff curves pertains to the whole system rather than isolated parts. Therefore the coordinates of one point on the surface can be used to determine from the nonlinear relationships the true cost of the system. By comparing the true costs for a number of points on the tradeoff curves, the optimum system can be found.

Even if the coefficients are assumed constant and one application of the synthesis procedure is sufficient to determine the optimum, it very well might be advisable to plot the the tradeoff curves to indicate where the optimum lies on the tradeoff surface. Slight modifications in the values assigned to the constant coefficients may result in considerably more palatable values for the parameters of the optimum network.

4. Setting Constant Coefficients

The values of the coefficients of the true cost polynomial will depend on the system being optimized and its application, so at best this section can give only a general qualitative appreciation of their determination. A thorough study should preceed the setting of coefficients to find all the factors which might conceivably bear on the cost of providing the system under study and the costs incurred should it fail. Some of these factors might well be intangible. An an aid to understanding the problems one must overcome while setting the coefficients, three examples are presented below.

Cost of Failure

Perhaps the most nebulous of all the coefficients of the true cost polynomial is the cost of failure. This is because many costs incurred in the event of a failure are intangible. For a military mission one of the more readily available costs is the cost of attempting the mission again if that can be done. The cost of abandoning the mission must also be included in many cases. This may well require the measurement of some intangible or difficult to estimate quantities. Questions will arise on the value of a human life or national prestige. Of course it is difficult to answer these questions but it may be attempted to determine a cost of failure.

The cost of failure should reflect only the costs incurred at the failure of the system under study. For instance, the system being optimized by the procedure may be one experiment of many carried in an earth satellite. Certainly the failure of this experiment causes a loss of valuable information. Its loss, however, is not the loss of the entire satellite since other experiments may still be operating.

On the other hand if the system under study is the equipment which encodes the information from all experiments for transmission to the earth, its failure will disable the link between satellite and the earth, completely aborting the mission. The cost of failure of this system should reflect the cost of the loss of the complete satellite with all its experiments.

Cost of Implementation

The cost of implementation should reflect both manufacturing and engineering. Different parts of the system may have different costs. Of particular interest to the designer whose goal is to find the optimum placement of restorers will be the cost of providing a restoring circuit.

Cost Per Pound of Weight

For a space mission this factor will probably include the expense of providing a booster with the power to lift one additional pound of load or the expense of reducing weight of the vehicle elsewhere by one pound. Generally this doesn't seem to be a linear factor. Different parts of the system may have different cost of weight coefficients.

Other factors are found in a similar manner. The factors that are included in the determination of the coefficients are really dependent on the system and its use. True, some of the coefficients are very difficult to determine precisely, but some attempt should be made so that one can obtain reasonable relationships between the parameters of the system. The synthesis procedure itself may point out gross errors in the values of the coefficients by yielding unreasonable values of the parameters. Such an occurence could indicate that the choice of coefficients has not been realistic and the coefficients should be reexamined. On the other hand, the occurence may also mean that the users concept of reasonable values of the parameters is in error, and this concept should be reexamined.

D. THE ISOLATING ARRAY SYNTHESIS PROCEDURE

It has been shown that whether the designer is faced with constant or non-constant coefficients, the determination of the optimum design rests on the ability to find the network which minimizes true cost for a set of constant coefficients. This is no small problem in itself. The most obvious approach to the solution is to try all alternatives, measuring their costs and picking out the most inexpensive design. Unfortunately, the number of alternatives one has to consider increases so rapidly with the size of the network that this approach is eminently impracticable for all but the smallest networks. For instance, a network with 100 locations has 2^{100} or about 10^{30} different network arrays. If with the aid of a high speed digital computer one could determine the cost of each alternative in a millisecond, he would be able to analyze 3.16×10^{10} alternatives per year. At this rate, it would take 3.16×10^{19} years to complete the synthesis procedure. This of course is an inordinate time.

Recognizing this exhaustive search approach as impracticable, the study has investigated several other approaches to the problem of synthesis. One of these, named "Isolating Array Synthesis Procedure" is the most promising.

The end product of the synthesis technique is ideally the one network array for which the True Cost of Section III. B. is minimized. The Isolating Array Synthesis Procedure tempers this goal somewhat by finding a design which minimizes a cost function which is an approximation to the True Cost. Its foremost advantage is that for most networks it will require far fewer calculations than the exhaustive search routine. The technique is deterministic in that at its conclusion the designer has one design which minimizes the cost function. This end result may very easily be the True Optimum, but since it is an approximation it may yield another network array which does not minimize the True Cost. The degree of deviation from the true optimum will be small, and will be the subject of future studies. The network resulting from the synthesis technique is called simply the optimum.

The following sections describe several considerations which are very important to the development and understanding of the synthesis procedure The operations described are the building blocks of the complete procedure.

1. The Effect of a Restorer

This section is included to give the reader some intuitive feel of the effects of restorers in a redundant system and of the utilization of these effects in the formulation of a synthesis procedure.

To illustrate the effect of a restorer, consider the shift register of figure 3-4, and assume the states of all the locations in the register except location 5 are specified as shown.

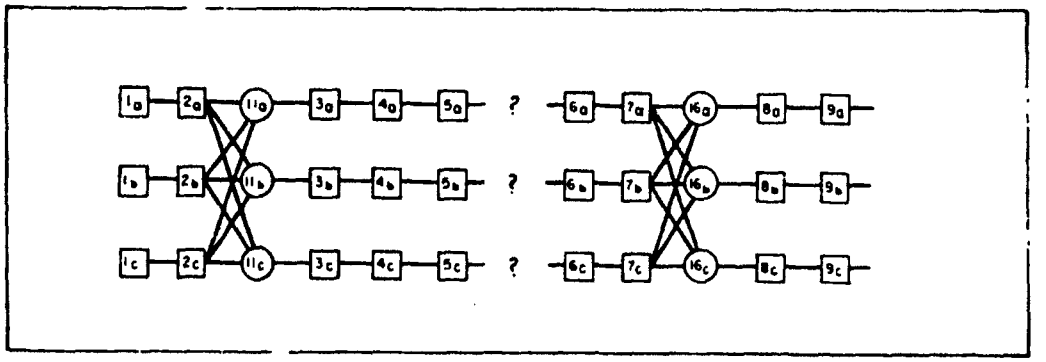


Figure 3-4. The Effect of a Restorer in a Shift Register

a. Reliability

The reliability of a network is the probability that at least k lines are successful at each network output. The addition of a restorer will, of course, change this probability. In a multiple line redundant network, in general, it takes more than one circuit failure to induce network failure. For the example, with majority restoring circuits, two properly placed circuit failures are required to disable the network. The causes of network failure can be divided into two classes: 1) the critical circuit failures all occur in the same function, (i.e. the failure of circuits 7a and 7b disable the network) and 2) the critical circuit failures occur in different functions, (i.e., 6a and 7b). The importance of this classification is that the addition of restorers can do nothing to reduce the first class but can reduce the number of combinations of failures in the second class.

Now, what is the effect, on reliability, of adding a restorer to a redundant-multiple-line network? Before the restorer is added, a list can be constructed which includes all the combinations of functions within which circuit failures can occur to cause the network to be disabled. For the example network with $x_5 = 0$, this list is shown in table 3-1. Entries with only one function describe combinations of the first class and entries with two functions describe combinations of the second class. If the order of redundancy of the example were greater, there would be entries with more than two functions. The number

associated with each entry is the number of different combinations of circuit failures that arise in the listed functions. For instance, there are three sets of two circuits in function 7 whose failure causes failure of the network, 7a-7b, 7a-7c, 7b-7c; and there are six sets of two failures in the functions 6 and 7, 7a-6b, 7a-6c, 7b-6a, 7b-6c, 7c-6a and 7c-6b.

Table 3-1. Combinations of Functions in Figure 3-4 with Location 5 empty in which Two Circuit Failures can occur to cause Network Failure

Combination of Functions or Restorers	Number of Fatal Combinations of Two Circuit Failures
1	3
2	3
3	3
4	3
5	3
6	3
7	3
8	3
9	3
11	3
16	3
1,2	6
11,3	6
11,4	6
11,5	6
11,6	6
11,7	6
3,4	6
3,5	6
3,6	6
3,7	6
4,5	6
4,6	6
4,7	6
5,6	6
5,7	6
6,7	6
16,8	6
16,9	6
8,9	6

A list such as the one in table 3-1 is important because it, together with the reliabilities of the circuits in the function and restorers describes an estimate of the reliability of the network. This estimate, which is described completely in Appendix A of the First Annual Report is called the Minimal Cut approximation to reliability. The Minimal Cut approximation gives a lower bound to the true reliability. It is quite accurate when the reliabilities of the circuits of the network are close to 1, say .99 or greater. This condition will be met for most networks for which the synthesis procedure will be used.

The reliability of the network calculated with minimal cuts is defined as the probability that none of the sets of circuits listed in table 3-1 fail. Two networks, with the same list, have the same reliability regardless of how the functions are interconnected. This approximation to reliability is used to determine the expected cost due to failure in the optimization criterion.

Then, assuming the circuit reliability in each function and restorer is known, using table 3-1, the reliability of the shift register in figure 3-4 can be calculated.

Now, when a restorer is added to location 5, a new list results. This is shown in table 3-2.

Table 3-2. Combination of Functions, in Figure 3-4 with Location 5 Filled, in which Two Circuit Failures can occur to cause Network Failure

Combination of Functions or Restorers	Number of Fatal Combinations of Two Circuit Failures
1	3
2	3
3	3
4	3
5	3
6	3
7	3
8	3
9	3
11	3
14	3
16	3
1,2	6
11,3	6
11,4	6
11,5	6
3,4	6
3,5	6
4,5	6
14,6	6
14,7	6
6,7	6
16,8	6
16,9	6
8,9	6

Table 3-3. Combinations Lost and Gained with the Addition of a Restorer in Location 5

Combinations Lost	Combinations Added
11,6 - 6	14 - 3
11,7 - 6	14,6 - 6
3,6 - 6	14,7 - 6
3,7 - 6	
4,6 - 6	
4,7 - 6	
5,6 - 6	
5,7 - 6	

Table 3-2 is different from table 3-1. Combinations have been gained and lost by the addition of the restorer. The gains and losses are summarized in table 3-3. When combinations are lost with none gained, the reliability of the network will always increase. However, when combinations are gained, with none lost, the reliability will always decrease. With the addition of the restorer, the network has both gained and lost circuit combinations whose failure brings about the network failure. It is not obvious, without calculating, whether the reliability has increased or decreased with the addition of the restorer. If the reliability for all circuits is the same, the number of combinations becomes the important parameter; the fewer failure inducing combinations, the greater the reliability. If this is the situation, for example, the restorer in location 5 is beneficial since its addition caused 48 combinations to be lost and 15 combinations to be gained.

How has all this come about? What mechanism has the restorer used to change the list of failure inducing circuit combinations? The answer to these questions can be seen in the error correcting properties of the restorer. From Section II.A., it is known that errors which appear on the input of a restorer and are insufficient in number to cause network failure are not passed through the restorer. As long as this condition holds, the number of errors on the output of the restorer is independent of the number of errors on its input. Restorer 11 and functions 3, 4 and 5 form the inputs to restorer 14 in location 5, and functions 6 and 7 are tied to its output. Since there is no signal path between members of the two sets of functions which bypasses the restorer, the restorer has isolated the effects of the circuit failures in 11, 3, 4 and 5 from circuit failures in 6 and 7. This is the reason for the restorer; it is the only beneficial effect inherent in its use.

The inclusion of the restorer has some effects on the reliability of the network that are not necessarily beneficial. Because the restorer is constructed of real physical restoring circuits, these circuits are necessarily subject to failure. Since these restoring

circuits were not in the network before the addition of the restorer, some new error inducing combinations are introduced with their inclusion. Of course, failure of two of the restoring circuits causes network failure; therefore, combinations of the first class (in the same function or restorer) are introduced. The restorer must take on all the outputs previously supplied by its function, so combinations of the second class (in two different functions or restorers) must also be introduced. Note that when a restorer is added to location 5, all the combinations consisting of functions 5 and its sinks (combinations 5,6 and 5,7) have been replaced by combinations of the restorer and those sinks (combinations 14,6 and 14,7). In general, when a restorer is placed in the location of a function, combinations including the restorer and the sinks of the function will always be gained. Combinations which include the function and its sinks will always be lost, unless, because of feedback in the network, the sinks in the combinations are also sources of the function.

The main point to be derived from this section is that the effect on the reliability of the network, due to a change in state of a particular location, is independent of some of the functions and the states of some of the locations of the network. Note that the combinations which include functions 1, 2, 8, 9 or restorer 16 do not change at all with the addition of the restorer in location 5. No combination lost or gained includes any of these functions or restorers; while all other functions and restorers in the network are included in one of the entries of table 3-3. As far as noting the difference in reliability between the networks with and without a restorer in location 5, these functions might just as well have been left out of the network and only the network of figure 3-5 considered.

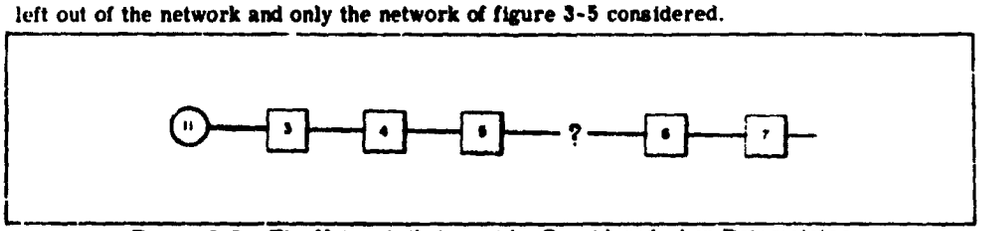


Figure 3-5. The Network that must be Considered when Determining the effect of a Restorer in Location 5

Since this small network need only be considered, it is reasonable to say that the effect of the state of location 5 on reliability is independent of the form of the network before restorer 11 or after function 7 as long as restorer 16 is present.

All this is so, because restorers 11 and 16 have isolated functions 5 from functions 1, 2, 8, 9, and restorer 16. There are no failure inducing combinations which include function 5 and any of these functions and restorers.

The network of figure 3-5 is called an isolated region of function 5. Every function or restorer not in the region is isolated from function 5, and every function or restorer within the region is error-linked to function 5. This region is described by an array called an isolating array of function 5, which specifies the states of locations on the boundaries of the region and within the region. For this example, the isolating array is (X, 1, 0, 0, 0, 0, 1, X, X). Locations 1, 8, and 9 are unspecified, X'd, because their states have no bearing on the effect of the state of location 5 on the reliability of the network.

b. Functional Cost

In the synthesis procedure, the decision whether to fill a location or leave it empty for a particular isolating array will depend on the functional cost of Section III. B., as well as the reliability. The effect on the functional cost of adding a restorer to location 5 is an obvious one. The restorer can in no way decrease or increase the costs of any other function or restorers in the network. It can only add on its own cost. If C_N is the cost of the network without the restorer in location 5 and C_R is the cost of the restorer, $C_N + C_R$ is the functional cost of the network with the restorer. A restorer can only increase the functional cost of a network, and the amount of increase is independent of the functions or interconnections of the network.

2. Determination of the Optimum State of the Location

The effects of adding a restorer to location 5 have been shown for the example. Now the problem is how to determine whether it is best to put a restorer in location 5 or leave it empty, the state of the other locations given.

First, if the designer is interested only in maximizing reliability, he will determine which state is more reliable and choose that one. It has already been indicated how this is done using the minimal cut approximation to reliability. It should be remembered here that when maximum reliability is the goal, the optimum state of location 5 does not depend on the functions or restorers which are isolated from function 5 when there is no restorer in that location. The state of location 5 should be set to maximize the reliability of its isolated region. Perhaps this fact is more easily accepted if it is remembered that if a network is made up of a number of independent parts, the maximum reliability of the network is obtained when the reliability of each part is maximized.

If the designer is interested only in minimizing the functional cost, he would leave the location empty regardless of the construction of the network, since the restorer only increases cost. Of course, if the designer is only interested in this parameter, he would not be using redundancy.

In the synthesis technique, both of these factors are considered in the optimization of the state of a location. They are used together in the True Cost equation.

For determining the optimum state of location 5, in the example, the True Cost* is calculated for the network which includes all the functions or restorers error-linked with function 5 when location 5 is empty. This network is the isolated region. The cost of this region is calculated first with location 5 empty and then with a restorer added to location 5. Both the functional cost and the expected cost due to failure will change with the addition of the restorer. The state of location 5 which has the least True Cost is judged the optimum for the array of states taken on by the other locations in the network, (X, 1, 0, 0, 0, 0, 1, X, X).

Since this comparison has been made not considering the form of the network beyond restorers 11 and 16, the decision on optimum state of location 5 is the same even though restorers may be added to locations 1, 8, and 9 or any set of these locations. Then this decision is good for any set of states taken on by locations 1, 8, and 9.

Table 3-4 lists the network arrays for which the optimum state of location 5 is determined by optimizing the location in the isolated region identified by the isolating array.

Table 3-4. Network Arrays in which the Optimum State of Location 5 is Determined by Optimizing the Location in the Isolated Region Identified by the Array (X, 1, 0, 0, 0, 0, 1, X, X)

(0, 1, 0, 0, 0, 0, 1, 0, 0)
(0, 1, 0, 0, 0, 0, 1, 0, 1)
1, 0, 0, 0, 0, 1, 1, 0)
(0, 1, 0, 0, 0, 0, 1, 1, 1)
(1, 1, 0, 0, 0, 0, 1, 0, 0)
(1, 1, 0, 0, 0, 0, 1, 0, 1)
(1, 1, 0, 0, 0, 0, 1, 1, 0)
(1, 1, 0, 0, 0, 0, 1, 1, 1)

The results of this section are extremely important to the synthesis procedure because it has shown how the principle of isolation has been used to optimize a location in a number of arrays through just two calculations of the True Cost.

* An approximation used in the procedure for the determination of this cost is described in Section III. F.

3. Optimizing a Location with Other Locations Already Optimized

It was described in the last section how a location within an isolated region can be optimized regardless of the states of locations outside the region. This section shows that a location can be optimized even though some of the other locations in the region are already optimized.

Consider the regions in figure 3-6 in which location 5 has been optimized by previous calculations for the two possible states of location 3. It is now desired to optimize location 3.

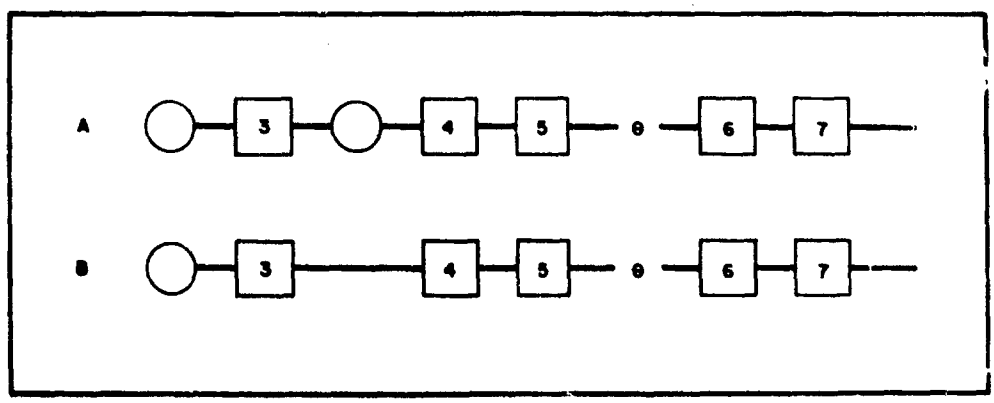


Figure 3-6. Optimizing Location 3 with Location 5 Already Optimized

The optimum state of a location will be either 1 or 0 depending on the results of some comparison of true costs, but usually in the examples of this report the fact that a location has been optimized will be indicated by a 0 which can take on either 1 or 0. This convention is used in figure 3-6. 0 indicates only that a location has been optimized. It in no way reflects the optimum state of a location. In fact if two locations are specified as 0 one may be restored while the other is not.

Note that the 0's in figure 3-6 may have been determined by the methods of the last section and that they may indeed be different. For a moment, assume they are the same. Then since the regions in "a" and "b" are identical and since location 3 is isolated by restorers in locations 2 and 7, the optimum state of location 3 can be determined by comparing the true costs of regions a and b. The optimum region is the one with the lowest true cost, and it is represented by the array, (X100001XX), in which locations 3 and 5 are optimized.

Now, what if the optimum state of location 5 were not identical for the two regions? The true costs of the regions "a" and "b" can still be compared to determine the optimum state of location 3. To illustrate this assume for location 5 that $\theta = 0$ for region "a" and $\theta = 1$ for region b.

With locations 2, 4, 6, & 7 specified as in figure 3-6 there are four possible configurations the region can assume as locations 3 and 5 take on all possible states. These are shown in figure 3-7 with their arrays.

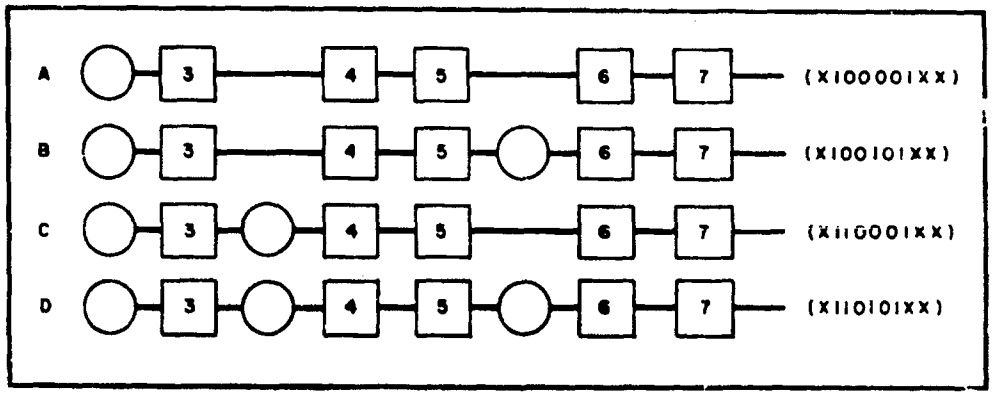


Figure 3-7. The Possible Regions as Locations 3 and 5 Assume All Possible States

From the previous comparisons one knows that with restorers in location 3 and 7 and none in locations 4 and 6, location 5 should be empty. Thus region c is less costly than region d, and the latter region has been eliminated by the previous comparison. Also from a previous comparison one knows that with restorers in 2 and 7 and none in locations 3, 4, & 6, location 5 should be filled. Thus alternative a has been eliminated in a previous comparison.

Thus to find the optimum state of location 3 with 5 already optimized, it is only necessary to compare regions b and c.

The result of the comparison will be an array with two locations optimized, (X100001XX). The values of the θ's are determined by the least expensive region. Thus if region b is least expensive, the θ in location 3 will be 0 and the θ in location 5 will be 1 If region c is least expensive, the θ in location 3 will be 1 and the θ in location 5 will be 0. These results are very important because they make it possible to optimize a location and use the results of the comparison to simplify subsequent determinations of the optimum state of other locations.

It is important that when two regions are compared to optimize a location that the same location be optimized in both regions. To illustrate the opposite consider figure 3-8.

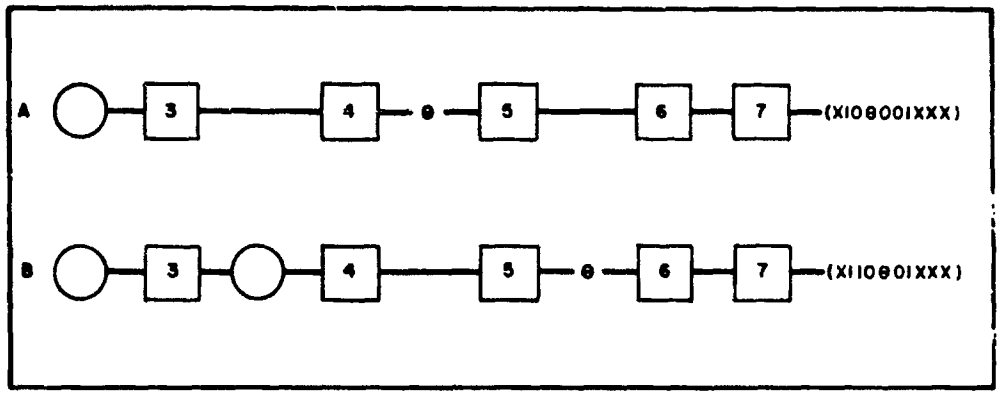


Figure 3-8. Two Regions With Different Locations Optimized

Comparison of these two arrays will of course determine which has the lowest true cost, but the question is what to do with the result. It cannot be said that the comparison yields the array, (X10001XXX), with three locations optimized, because not all the alternative states of the three locations have been considered. For instance, assume that the 0's in both regions a and b are equal to 1. Of the eight possible regions that occur when locations 3, 4, and 5 assume all possible values shown in figure 3-9, only two, a and e have been eliminated by previous comparisons.

The results of the comparison of the regions in figure 3-8 will yield either f or c. Thus four of the eight alternatives have not even been considered, hence the comparison cannot yield the optimum values of locations 3, 4, and 5.

The findings of this section can be stated in general. Two arrays can be compared to optimize a location in spite of the presence of previously optimized locations. However, the same locations must be optimized in each array. The results of the comparison will be an array which is optimized for all locations optimized before the comparison plus the location optimized by the comparison. The values of the 0's in the new array will be the states of the corresponding locations in the least expensive of the two regions in the comparison.

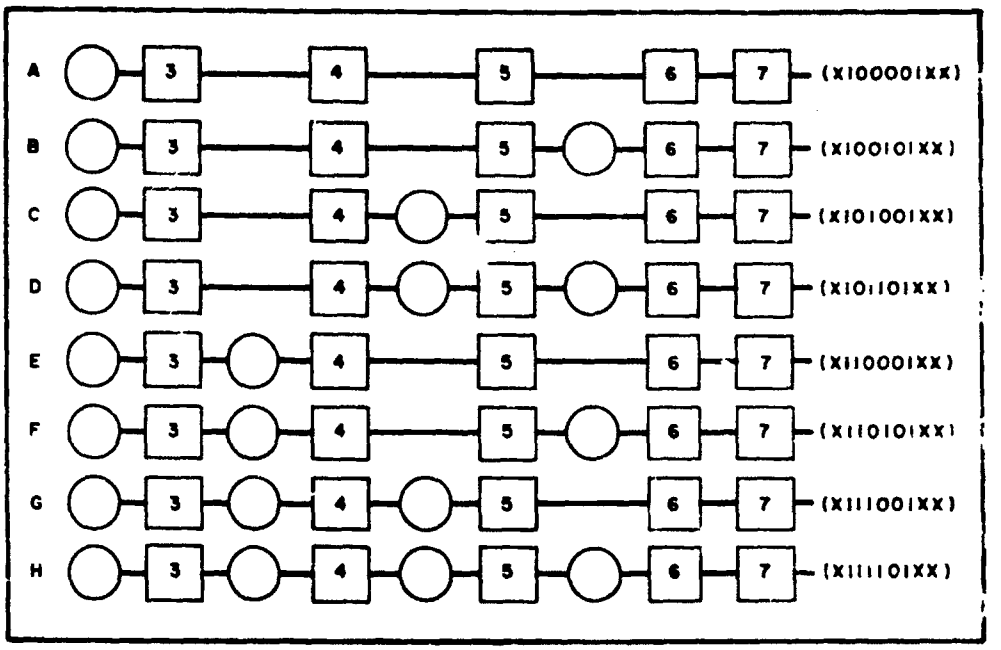


Figure 3-9. The Possible Regions as Locations 3, 4, and 5 Assume All Possible States

4. Comparison of Two Arrays to Optimize a Location

In the synthesis procedure to be described in the following pages a frequently referenced operation will be the comparison of two arrays to optimize a location. This section is devoted to the description of this operation.

Two arrays are said to be comparable if there is one and only one location which takes on the "1" state in one of the arrays and the "0" state in the other and every location specified as 0 in one array is also specified as 0 in the other array. For example the arrays

(1 0 1 1 X X 0 X 0)

and

(1 0 0 1 X X 0 1 0)

are comparable. On the other hand, the arrays:

(1 0 1 0 X X 0 X 0)

and

(1 0 0 1 X X 0 1 0)

or the arrays:

(1 0 1 1 X X X X 0)

(1 0 0 1 X X 0 1 0)

are not comparable.

Comparable arrays can be used to derive two regions which are identical except for one location and possibly some previously optimized locations; hence the comparison of comparable arrays (or alternatively the comparisons of the regions) yields an array with one more location optimized.

Since an array may have a 1 or 0 where a comparable array has an X, regions specified by comparable arrays may not be identical. For instance for a nine function shift register two comparable arrays are:

(1 X X X X X X X X)

(0 0 0 1 X X X X)

These arrays form the regions shown in figure 3-10, a and 3-10, b respectively.

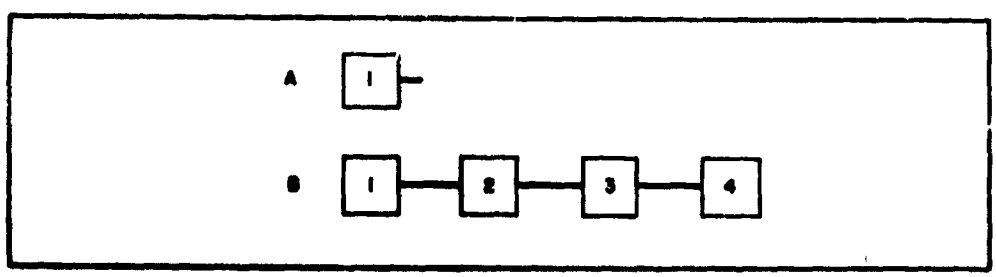


Figure 3-10. The Regions of Two Comparable Arrays

Although the arrays fulfill the requirements for comparability their regions cannot be compared because they do not include the same functions. To form the regions which are to be compared to optimize a location, the regions will be made to include every function that is in either region specified by the arrays. Thus the regions of figure 3-10 will be modified to appear as in figure 3-11.

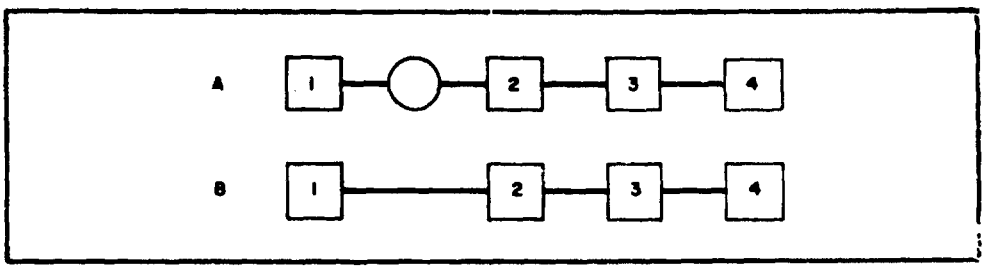


Figure 3-11. Regions Modified to be Comparable

Now the true costs of the two regions of figure 3-11 can be compared to optimize the state of location 1 for the given values of the other locations.

In summary comparable arrays differ in one and only one location such that a 1 appears in one of the arrays and a 0 appears in the other. Excluding the differing location any location which is specified as 0 in one array must be 0 or X in the other array and any location specified as 1 in one array must specify as 1 or X in the other. Before two arrays can be compared to optimize a location, the regions must be made to include the same functions. Every function included in either of the regions of the arrays must be included in both modified regions.

E. THE DETAILED SYNTHESIS PROCEDURE

The procedure described in this section differs in one basic respect from the procedure described in the First Annual Report to perform the same function. The First Annual Report's procedure generated all the isolating arrays of function 1 and placed them in a list. It then searched through the list to find the proper arrays to compare in order to optimize locations. The current version, here described, generates the arrays in the order they are needed, and makes all possible comparisons with an array before another array is generated. The long list of arrays is no longer necessary, hence the memory space and search time required for the computer implementation of the procedure are greatly reduced. Major portions of the new procedure are the techniques required to generate arrays in the proper order and to recognize the proper arrays to be compared.

This section describes all the steps of the Isolating Array Synthesis Procedures. The activities of the procedures can be divided into parts, those dealing with: 1. The generation of the isolating arrays in the proper order and 2. The comparisons of the arrays to optimize locations. In the procedure the generation and comparison activities are performed concurrently but for this description they will be treated separately.

From time to time names are introduced into the text which are variables in the program which implements the procedure on a digital computer. These are introduced to ease the description of parts of the procedure, rather than to illustrate the construction of the program. A description of the program is reserved for Appendix B.

1. The Generation Procedure

a. Terminology and Background

(1) General

The problem of generating the isolating arrays is equivalent to the problem of generating the isolated region which they represent. To help visualize the process used to generate these arrays, think of the generation procedure as constructing a pod, which, at any given time, encloses a part of the network which includes function 1. This enclosed part of the network is isolated by restorers or the form of the network from the remainder of the network and forms an isolated region which is independent of the remainder of the network. Every function and restorer within the pod is to be error linked to the first function. The generation process begins as an enclosure about a single function (function 1) in the network and as it develops, the area enclosed by the pod increases until, finally the entire network is engulfed by the pod.

(2) Incompletely Specified Arrays

The boundaries of the area enclosed by the pod are locations in the network and they are referred to in the isolating arrays as δ 's and $\emptyset$'s. These designations are variables and will take on the states 1 and 0 - i. e. the location described by the δ or $\emptyset$ will be allowed to first be restored, then be unrestored. When one location with δ or $\emptyset$ is allowed to vary two new arrays are generated, the 1-substituted and the 0-substituted arrays. Arrays which include one or more δ 's or $\emptyset$'s are referred to as incompletely specified arrays. They are intermediate, or transition, arrays in the generation procedure. They eventually give rise to completely specified arrays (arrays which contain no δ 's or $\emptyset$'s). The generation of completely specified arrays is the purpose of the generation procedure.

(3) The Implications of δ and $\emptyset$ Variation

As previously discussed, an imaginary pod is constructed about a given function in the network and this pod is allowed to expand until the entire network is encompassed. If this first function is selected at some place in the middle of the network this expansion of the pod can take two directions with respect to the signal flow: upstream or downstream from the function selected. The difference in the effects experienced lies in the manner in which error-linking occurs. When the expansion occurs in the upstream direction the first type of error linking, the upstream type, occurs, hence upstream boundaries at the pod are specified as $\emptyset$. Conversely, when the expansion is proceeding in the downstream direction, downstream error-linking is possible, hence δ specifies the downstream boundaries of the pod. If the δ's and $\emptyset$'s were replaced by 1's, the enclosed area would form an isolated region of function 1.

(4) Effects of δ and $\emptyset$ Variation

δ and $\emptyset$ are variable states for a given location. Their presence in an array means that the array is incompletely specified and that more arrays will be generated from this array. One at a time each δ and $\emptyset$ will take on the two possible states 1 and 0. Hence, the variation of a δ or $\emptyset$ yields two arrays. The 1-substituted and the 0-subetituted arrays. When a 1 is substituted for a δ or $\emptyset$, a restorer is assumed in that location and no error-linked functions are added. However, when a 0 is substituted, the absence of a restorer is assumed and error-linking takes place between the function whose output was the δ or $\emptyset$ and its error-linked functions. The manner in which error-linked functions are added is determined by whether a δ or $\emptyset$ was varied. Varying a δ adds error-linked functions according to the downstream effect whereas a $\emptyset$ adds error-linked functions according to the upstream effect. Each newly added error-linked function's location takes on the variable designation δ or $\emptyset$ depending upon whether it was added while moving

upstream or downstream. If it was come upon while moving downstream a δ is added, if upstream a $\emptyset$ is added in the proper location. $\emptyset$'s are added due to upstream error-linking while δ and $\emptyset$'s may be added due to downstream error-linking.

(5) The Parent Array

In order to initiate the generation procedure an incompletely specified array must be constructed which builds this imaginary pod about the function selected as function 1 in the network. This is accomplished by assuming the output of this function to be a δ and any input error-linked functions as $\emptyset$'s. Since this is the initial array, that which gives birth to all others, it is referred to as the parent array. Allowing the δ and $\emptyset$'s to take on their variable states allows the pod to grow and encompass an increasing number of functions.

(6) Example

Figure 3-12 a. shows a network and its parent array. The imaginary pod encloses only function 1, its output (downstream) is specified as δ, its input (upstream) is specified as $\emptyset$. Figure 3-12 b. represents the result when the δ in location 1 is allowed to vary - the two arrays shown result. The 1-substituted array (1$\emptyset$ xxxxx) adds no new error-linked function. The 0-substituted array (0$\emptyset$ x x $\emptyset$ δ x) has added two new error-linked functions due to the downstream effect. If no restorer is assumed in location 1, function 1 is error-linked to function 6. Function 6 was found by moving downstream from function 1, hence, it is specified as δ and may introduce more downstream effects when it is allowed to take on the 0 state at a later time. Function 5 was found by moving upstream from the function which has inputs from two branches. When this location is allowed to take on the two states, a 0-substitution should introduce only upstream effects, hence it is designated $\emptyset$. The new pod encloses function 1 and 6. This occurs when the variable locations 2, 5 and 6 have taken on the 1 state simultaneously. The pod has expanded in the downstream direction.

b. Mechanics of the Procedure

The aim of the generation procedure is to arrive at all the completely specified (no δ's or $\emptyset$'s) isolating arrays of function 1. In addition it is necessary to make the proper comparisons between these isolating arrays in order to optimize the states of the locations of the network.

As the pod about the first function is increased in size, not only are isolating arrays of function 1 generated, but other functions are incorporated such that isolating arrays of these functions are also being generated. Hence, comparisons can be made to optimize these other locations in addition to location 1. The method which is used to generate and compare these arrays will be discussed here

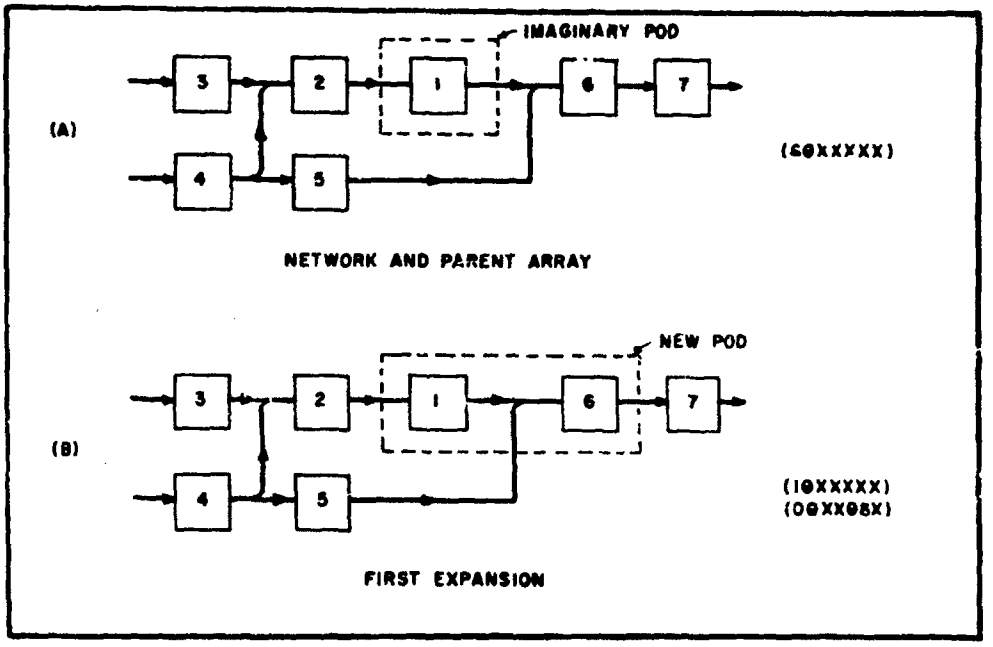


Figure 3-12. Example Network and Arrays Generated by First Expansion of the Pod

All isolating arrays generated by the synthesis procedure are derived from the parent array. The parent array normally contains a number of variable designations ($\emptyset$'s or $\not\emptyset$'s). When a substitution is made for a $\emptyset$ or $\not\emptyset$, two arrays result, one with a 1 substituted for the $\emptyset$ or $\not\emptyset$, the other with a 0 substituted. Both of these arrays will contain all the remaining $\emptyset$'s and $\not\emptyset$'s from the parent array. In addition the 0-substituted array may have added more error-linked functions ($\emptyset$'s or $\not\emptyset$'s). The parent array and its two derivations are shown in Figure 3-12 for the network shown.

If both arrays generated are incompletely specified the 0-substituted array is temporarily stored* and the procedure operates on the 1-substituted array, varying the next $\emptyset$ or $\not\emptyset$. The procedure continues to operate on the derivations of the 1-substituted array until all completely specified arrays result and all those comparisons which are necessary at

* In the program implementing the synthesis procedure, a list is maintained of incompletely specified arrays awaiting processing. When an array is processed it is removed from the list. This list is called INCAR. In subsequent discussions this name is used to describe the list.

this point have been carried out. A more detailed analysis of the comparisons which are made will be presented in a subsequent discussion**. Then the 0-substituted derivative of the parent array is operated upon.

c. Branch Effects

(1) Branch Formation

If this 1-substituted array should give rise to two additional incompletely specified arrays, again the 0-substituted of the two is temporarily stored and the 1-substituted is operated upon. This type of procedure leads to the formation of "branches" of arrays. It can also be true that a 1 substituted array may be completely specified while its partner, the 0-substituted array, is incompletely specified. If this is true the 1-substituted array is carried through as many comparisons as possible (this will be further explained later) and the 0-substituted array is then operated upon to generate additional arrays.

(2) Branch Ends

A Branch End is encountered when the two arrays generated from an incompletely specified array are both completely specified. This halts the generation process for this branch and after the necessary comparisons are finished, the next branch is considered.

The next branch is begun by locating the last incompletely specified array to be temporarily stored: (In INCAR). Operation on this array yields the next branch. This process continues until there are no remaining incompletely specified arrays in storage. This means that the last branch has been developed and all possible isolating arrays of the first location have been generated.

2. The Comparison Process

a. Comparison Within a Branch

A typical branch which might be developed in the generator procedure is shown in figure 3-13. This branch contains several completely specified arrays as well as a number of transitory incompletely specified arrays.

** The concept of comparison is introduced here to give the reader a feel for the contribution of the generation process to the comparison and optimization process.

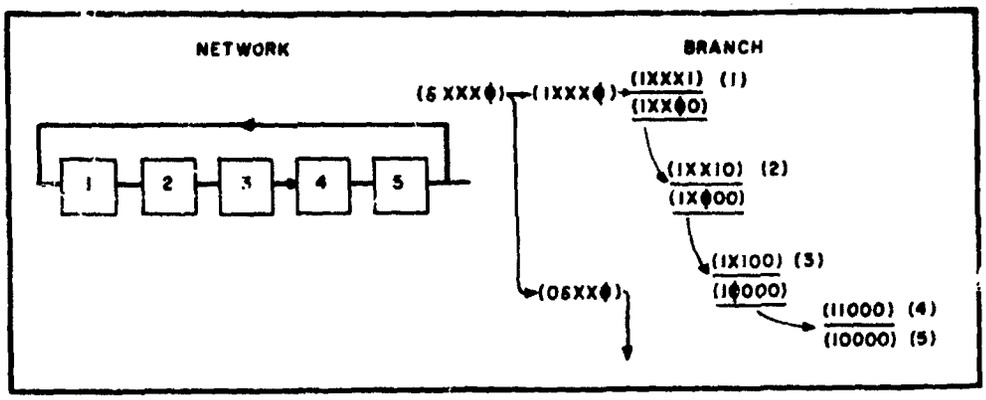


Figure 3-13. Example Showing Optimization of Locations Within a Branch

Any completely specified array in the branch differs from each of those which follow it in the same branch by only one specified location, i.e. there is a location in the array which is in the 1 state which in each of the arrays following is in the 0 state. This is due to the manner in which the arrays are generated. When this location was varied, first a 1 was substituted, then a 0. The 1-substituted array was completely specified; the 0-substituted array was incompletely specified, leading to additional arrays. Those following arrays have additional 1's and 0's where the 1-substituted array contains X's. These additional 1's and 0's were introduced due to the δ's and ϕ's which were introduced by the 0-substitution. However these locations were not specified as 1 or 0 in the first array.

Notice, in Figures 3-13, array 7 has only 2 specified locations, 1 and 5. Each of the arrays following in the branch differ from array 1 in location 5*.

Physically, array 1 (figure 3-13) represents an isolated region of function ' as do the other completely specified arrays. If the region which is included in array 7 but not in array 1 is added to the region for array 1, the two resulting regions will differ in only one location. Comparison of these two arrays will yield an array with one location optimized, that location by which the two arrays differ. Likewise, comparison of array 1 with each of the others will yield arrays with one location optimized. (See figure 3-14a).

* The example of figure 3-13 illustrates the format in which a branch is written. The arrows indicate two arrays which are generated when a variable is allowed to take on its two states. This format is employed to help the analyst identify those comparisons which must be performed.

In general an array is compared with each array generated before it which differs from it in only one specified location. This criterion can easily be seen by use of the branch technique since any two arrays which are in the same branch differ from each other in only one specified location. Hence, by examining the entire generation true for a network the comparisons which will be made will be evident.

array	result
2	(1XX10)
3	(1X100)
4	(11000)
5	(10000)

a. Results of the first comparison; array 1 with arrays 2, 3, 4 and 5.

array	result
3	(1X100)
4	(11000)
5	(10000)

b. Results of second comparison - array 2 with 3, 4 and 5

array	result
4	(11000)
5	(10000)

c. Results of third comparison - array 3 with 4 and 5

array	result
5	(10000)

d. Results of fourth comparison - array 4 with 5

Figure 3-14. Results of First, Second, Third, and Fourth Comparisons

Since array 1 differs from each of these generated after it in the same location, the optimized location will be the same for each comparison. Now if array 2 is compared with each of those following, another location will be optimized in arrays 3, 4, and 5.

Successive comparisons of 3 with 4 and 5, and 4 with 5 will finally yield an array with four locations optimized. These four optimized locations are optimum under the conditions of array 5, i.e. if the restorer configuration of array 5 (a restorer in location 1) exists the four optimized locations represent the best possible configuration of restorers for the network.

b. Comparison Between Two Branches

The result of comparisons in a given branch yields, in general, an array with x number of locations optimized for a given restorer configuration. The next branch generated in the procedure will, likewise, produce an array with x number of locations optimized, but for a different configuration of restorers. Figure 3-15 shows such a condition.

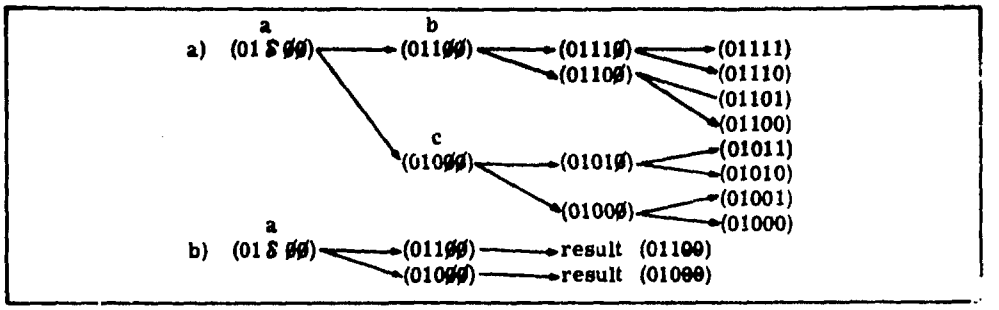


Figure 3-15. Example of Branches and Partially Optimized Arrays
a. The branches for an example
b. The partially optimized arrays that arise in the example

Given the system of arrays shown in figure 3-15, arrays b and c were generated from array a and, hence, differ from each other in only one location, that which was varied. Each gives rise to a branch.

These two branches must differ in only this one specified location. Therefore, after the comparisons are made for each branch, two arrays result each of which contains two optimized locations each for a given configuration of restorers shown in figure 3-15b. These two arrays may now be compared to optimize yet another location. These three locations are now optimized under the restraint of the configuration of restorers described by the remaining specified locations in the resulting array. Now, however, there is one less specified, non-optimized location than there was in each of the arrays which had two locations optimized. In general, successive comparisons between "branch results" increase the number of optimized locations and decrease the restraints until, finally, these result in a comparison between two arrays each of which has all but one location optimized and which differ in this one location. Comparison of these two arrays yields the optimized network.

c. Mechanics of the Comparison Process

In order to avoid the storage and manipulation of a large number of completely specified arrays, each array is compared and optimized as many times as possible immediately after its generation. The comparison rule may be stated as follows: A completely specified array is compared immediately with that array in storage which has the same number of locations optimized.

The first completely specified array generated will represent the smallest possible isolated region which includes the first function. It is a result of 1's being substituted for all the δ's and φ's in the parent array. The next completely specified array generated will represent a slightly larger region and will differ from the first in one specified location. An immediate comparison can be made to optimize this location subject to the restraints of the 1's in the second array.

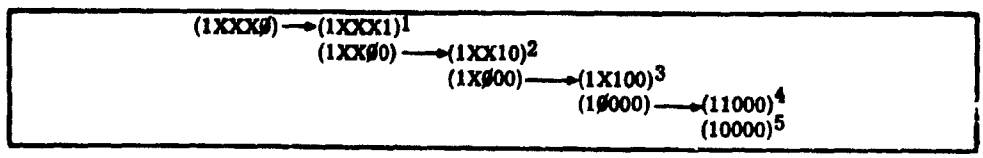


Figure 3-16. Successive Comparisons

Refering to figure 3-16, upon generation of array 2 an immediate comparison may be made between arrays 1 and 2. This is done even before the remainder of the arrays are generated. This results in array 2 with one location optimized (1XX10)*. The third array generated (initially no locations optimized) may be compared immediately with the first array which also has no locations optimized. This comparison results in one optimized location (location 5). This is the same location as the previous comparison optimized since both arrays differ from the first array in this same location. Now the second and third arrays may be compared since each is optimized in one location. This comparison results in a second optimized location (location 4). This progression is not unlike building a staircase - each new array must complete all the comparisons of its predecessor before it can optimize an additional location. This progression continues until the end of a branch is reached. The final array, optimized as much as possible, contains the most optimized locations and the fewest restraints in the branch. When the end of the branch is reached and this array with the most locations optimized is obtained, all the other arrays that were generated in the branch must be dropped from consideration. Only the array with the most locations optimized is retained. This step assures that no two partially optimized arrays have the same number of locations optimized.

For the example, when array 5 is generated it has no optimized locations and therefore compares to array 1. This yields one optimized location (10000). Comparison is then made with array 2, which also has one location optimized, yielding (10000). After successive comparisons with 3 and 4 which have respectively 2 and 3 locations optimized the resulting array is (1). The same result was noted in figure 3-14.

As the next branch is generated the completely specified arrays which are generated in it may be compared only with arrays which differ from them in one specified non-optimized location.

---

* In the procedure partially optimized completely specified arrays are stored in a memory location called KOMPAR. Along with this array is a list, KLO, which indicates the number of locations optimized in each entry in KOMPAR.

The initial array in a new branch is a 0-substituted array which resulted from the 1 and 0 substitution for a $ or ø. The 1-substituted array formed the previous branch. This 0-substituted array differs from its partner (the 1-substituted array) and any predecessor in only one specified location. Any other arrays generated from the 1-substituted (partner) array will differ from the 0-substituted array in more than one specified non-optimized location. Hence, comparison between members of two different branches is invalid except for the one final array which has the greatest number of locations optimized. This one array is a representative of the entire branch for successive comparisons. Every location which was varied after the formation of the 1-substituted array which gave rise to the branch, was later optimized. It may be thought of as an optimized, 1-substituted array and it differs from the 0-substituted array and all its derivatives by only one non-optimized location. The completely specified arrays generated from the 0-substituted array compare and optimize locations in the staircase manner as did the previous branch until the same number of locations are optimized as the previous branch. Then comparison is made between the results for the two branches, and an additional location is optimized. The result is the array which represents the optimum case for a larger branch (that made up of two smaller branches). The next branch initiates a new staircase until the previously high number of locations optimized is reached where upon comparison can be made between branches. This process continues until all locations but one are optimized for two arrays and a final comparison between them is made. The result of this comparison is the optimum network.

3. The Order in Which $'s and ø's are Varied

The generation of arrays during the synthesis procedure is accomplished by letting the $'s and ø's in incompletely specified arrays assume the 1 and 0 state. At any one point in the procedure, only one $ or ø is to be varied at a time and the process generates two arrays, the 1-specified array and the 0-specified array.

When one has an incompletely specified array with several $'s and ø's, he must determine the proper one to vary. The successful completion of the procedure depends on the proper choice.

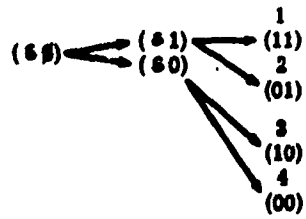
Consider the point in the procedure in which a 0-specified array called A has had as many locations optimized as possible. This array is to be placed in KOMPAR and a new incompletely specified array called B is to be chosen from INCAR to generate a new branch. From the way in which the synthesis procedure is performed, one knows that there will be one and only one location which is specified as 1 in array A and specified as 0 in array B. Call this location y.

During the optimizations that take place in the branch derived from the array B, there will arise one or more completely specified arrays called C which have the same number of locations optimized as A. When this occurs, arrays A and C will be compared, optimizing location y. As described in Section III. D. 2, for this comparison to be a legitimate one, array C must have optimized exactly the same locations as array A. This happy result is assured if the δ's and ϕ's are varied in the correct order in the branch derived from array B.

To determine the correct order of variation, consider the goal of having exactly the same locations optimized in C as are optimized in A. This will be assured if the first locations optimized in the branch derived from array B are exactly those locations optimized in array A. The order in which the δ's and ϕ's are varied determines which locations will be optimized first. It happens that if there is more than one δ or ϕ in an array, the location varied first is optimized last, the location varied second is optimized the second from the last and so on. For a very simple example of this, consider figure 3-17.

Figure 3-17. A Simple Example to Illustrate the Variation of δ's and ϕ's

Varying the ϕ in 2 and the δ in 1 in that order, one obtains the tree:



By comparing arrays (1) and (2), one optimizes location 1 yielding the array (0 1). By comparing arrays (3) and (4), one optimizes location 1 yielding (0 0). Comparing the arrays (0 1) and (0 0) optimizes location 2. By varying locations in the order 2 - 1, the locations have been optimized in the order 1 - 2. If the order of variation had been 1 - 2, the order of optimization would have been 2 - 1.

Stating the forgoing in a more formal manner, say that in array A the set of optimized location is α, and in array B the set of locations which are δ's or ϕ's is ρ. The rule for the order in which the locations in the set ρ are varied is: The δ's or ϕ's in locations in the set ρ but not in the α should be varied before the δ's and ϕ's in both the

sets σ and ρ . Following this rule assures that the locations optimized in A will be optimized first in the branch generated from B so that the proper comparisons can be made.

The mechanism, used in the synthesis procedure for choosing the correct δ or ϕ to vary, has been made somewhat more restricted than the general rule to simplify automation. It involves the use of an array in which the last 0-specified array to be optimized is stored (the last array optimized before a new branch is originated). In the program this array is called IPOPAR. This name is used in the following discussion.

The optimized locations in IPOPAR are indicated by θ_0's and θ_1's. The incompletely specified array from which the new branch is to be generated is scanned from the left for δ's or ϕ's. When one is found, the corresponding location in IPOPAR is tested to see if it is a θ_0 or a θ_1. If it is one of these values, the scan continues to the right until another δ or ϕ is found and the test repeated. If the location in IPOPAR is neither θ_1 or θ_0, the δ or ϕ in that location of the incompletely specified array is varied and the synthesis procedure continues to the next step.

If all of the δ's and ϕ's in the incompletely specified array are 0's in IPOPAR, the procedure varies the left most one.

4. Test to Determine if a Function is Isolated

An important condition which must be met before a location can be optimized is that the location's function must be isolated. Calling the location to be optimized k and its function j, this means that with location k empty, the function j must not be error-linked to any function whose location is an X. To illustrate a case where this condition is not met, consider the network in figure 3-18.

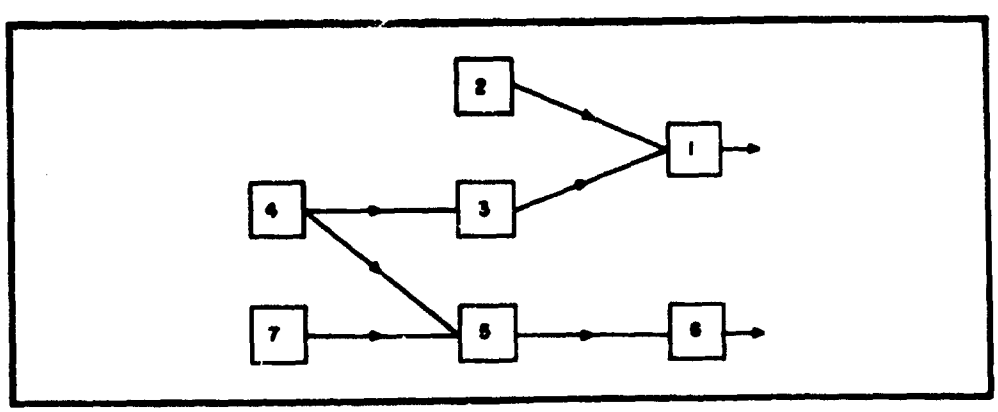


Figure 3-18. Network in Which Isolating Array of Function 1 is not Necessarily an Isolating Array of Function 4

Two isolating arrays of function 1 are the arrays:

(0000XXX)

and (0001XXX)

Function 5, 6, and 7 in both these arrays are isolated from function 1 by the form of the network. At first glance, these two arrays appear to be comparable; the result being an array with location 4 optimized: (0009XX). This is not true, however, because the optimum state of location 4 may depend on the states of locations 5, 6, and 7. Certainly one can see that the decision on the optimum state of location 4 depends on whether or not locations 5 or 7 include a restorer. If location 5 is empty the decision also depends on whether or not location 6 includes a restorer.

Such a condition may easily arise in the synthesis procedure, because the generation procedure creates arrays which isolate function 1, but there is no guarantee that they isolate any other function.

To allow for this occurence, the procedure tests to see if a location is isolated before it optimizes it. For example, say that the two arrays, IWORKG and JARC*, each with three locations optimized, are to be compared:

IWORKG = (0 0 0 0 X X X)

JARC = (0 0 0 1 X X X)

The optimization of locations 1, 2, and 3 are acceptable, because an array which isolates function 1 and specifies location 2 as 0 also isolates function 2. In like manner function 3 is isolated by an array which isolates function 1 and specifies location 3 as 0. To determine if location 4 can be optimized the procedure tests the array IWORKG to see if it isolates function 4. It finds that location 4 cannot be optimized.

A function is tested for isolation by checking to see if all of the locations of its primary sinks and all of the locations of the primary sources of these sinks are specified by 0, 1, θ_1, or θ_2 in the array. If such is the case, one can be sure the function is isolated. If such is not the case and some of these locations are specified as X's, the function is not isolated and the comparison cannot be made. For the example function 5 is a primary sink of function 4 and function 7 is a primary source of this sink and both locations are X's. Therefore, function 4 is not isolated.

---

* These are the computer program names given to arrays that are to be compared. They are used here simply to aid in the description.

When the condition of isolation is not met, the array must be modified so that the function becomes isolated. This is done by placing δ 's on all the primary sinks of the function whose locations are X, and ∅'s on all the primary sources of these sinks whose locations are X. For the example array this yields: (0000 δX∅).

This type of array is called an Indirectly Incompletely Specified Array to indicate the indirect manner in which the array became incompletely specified. The array is treated like a normal incompletely specified array and is placed in INCAR for further processing. As the δ's and ∅'s take on the 1 or 0 state in subsequent processing function 4 will be isolated. Note that these arrays may include optimized locations.

The synthesis procedure subjects every function to a test for isolation before its location is optimized.

5. Link-Limit to Simplify the Procedure

Even through the synthesis procedure is designed to require a far lower number of calculations than the exhaustive search procedure, for large networks the time required for synthesis may still be excessive. There is a rule of thumb, however, which can be used in the procedure to considerably reduce the number of arrays generated and the number of calculations made. This rule is called the link-limit.

The link-limit states that there will be a limited number of functions error linked to any restored function. Since the benefits of redundancy depend to a considerable extent on the presence of restorers in the network, the optimum design of any network will probably include a number of restorers. During the synthesis procedure comparisons between network designs in which most of the location are empty probably do not contribute to finding the optimum. The use of the link-limit eliminates from consideration most of these unnecessary comparisons.

The simplification requires the introduction of a quantity JTHLD. * This quantity is the maximum number of error-linked functions which all play a part in providing the input to a restorer. Thus JTHLD-1 is the maximum number of error-linked sources a restored function may have. The quantity JTHLD is fixed by the designer and represents what he feels to be a reasonable limit. Considerations on the setting of the quantity will be presented later in this section. The link-limit eliminates from consideration every array for which the number of error-linked sources of any of its restorers functions exceeds the limit. These arrays are called implausible. Figure 3-19, a and c represent arrays which are implausible when JTHLD equals 3. Figure 3-19, b represents an array which is plausible.

* Once again this is program terminology introduced for simplicity.

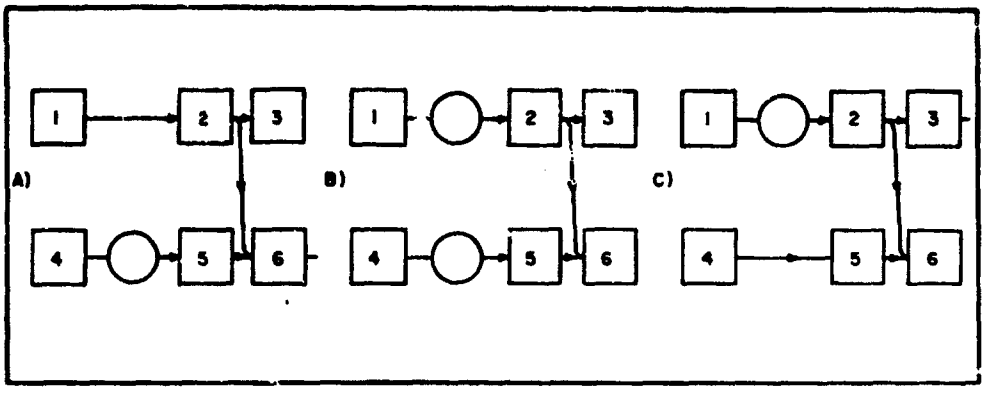


Figure 3-19. Plausible and Implausible Arrays

In every application of the synthesis procedure for a large network, many arrays will arise which are implausible. Elimination of these arrays from the synthesis procedure will considerably reduce the number of calculations which will be required for synthesis. With the link-limit savings are introduced by two mechanisms.

First, each array is tested for plausibility as it is compared with other arrays to optimize locations. As an array, IWORKG, enters the comparison procedure with no locations optimized, an array is found in KOMPAR with no locations optimized (if one exists). This array is set equal to JARC. IWORKG is tested for plausibility and if it is found implausible, JARC is called the best of the two arrays without any further calculations. JARC necessarily has one more 1 (restorer) than IWORKG so it is more likely plausible. One location has now been optimized in JARC and the array is now set equal to IWORKG for another pass through the comparison process and another plausibility test. By finding the array implausible, two cost determinations have been eliminated. It is unnecessary to determine the costs of either IWORKG or JARC since IWORKG is implausible. The plausibility test continues at each pass through the comparison process until a plausible array is found, then the synthesis procedes normally.

If IWORKG is found plausible the comparison procedes normally. No further plausibility tests are made on this array as more locations are optimized. Subsequent optimizations can only add more restorers to the network, hence increase the array's chances for plausibility.

An an example of the forgoing considerations consider the array A, in figure 3-20 which is to be compared with arrays B, C, D and E which are stored in KOMPAR.

become 1 as the branch is generated from the incompletely specified array. After this is done,the locations which are 0's in the new modified array will be 0 in every array which will be derived from the incompletely specified array being tested.

When it is determined that every array that will be generated from an incompletely specified array is implausible, the array is called an impossible array.

For simplicity call the array drawn from INCAR A, and the modified test array B. To test the possibility of A only locations which are 0 for all members of the branch will be made 0 for the test array B. To assure that this occurs, 1's will be assigned to B in two ways. First the locations which are 1's, 3's, or ∅'s in A are made 1 in B. All these locations will assume the 1 state in some array generated in the branch.

Secondly 1's will be added to some of the locations which are to be optimized in the branch. Generally whenever a branch is being worked on, there will be some list of partially optimized arrays in KOMPAR. Each member of this list will have some number of locations optimized, and no two members will have the same number optimized. If one were to look down the list of number of locations optimized, KLO, he might find something like; 0, 1, 2, 5, 6, 9. The list in general will not include every integer from zero to some maximum number, but will contain one or more gaps. While working on this branch, the lowest gap will be filled. First an array with three locations optimized will be generated and then an array with four locations. The last array to be generated in the branch will be compared with every array is the list of partially optimized arrays until an array is found with one more location optimized than the greatest consecutive integer in the list. The purpose of the branch generated from array A is to generate an array with the same number of locations optimized as the array in KOMPAR with the smallest number of locations optimized above the gap. Thus for the example the branch is to generate an array with 5 locations optimized. Call this array C. The locations optimized in C are the locations to be made 1's in the test array B.

Now there are some zeros still remaining in array B. Every array generated from A will have at least this many zeros. Thus if the array B is tested for plausibility and found implausible one knows that every array in the branch will be implausible. Array A is then judged impossible.

The impossible array cannot be thrown out because some of its 3's or ∅'s may open paths into parts of the network which can not be reached in any other manner. Many of the calculations that are made in the branch can be discarded however, because if the procedure were continued in the normal manner, all comparisons made in the branch will be between implausible arrays.

The unnecessary comparisons are eliminated by constructing a new array D by combining the arrays A and C. The array D is formed by replacing with one exception the contents of every location in array A with the contents of the locations which are specified as 1, 0, θ_1, or θ_0 in array C. Locations specified as X in C take on the states specified in A. The one exception is the location which would have been optimized by comparing an array generated in the branch with array C. Call this location y. Location y will have been 0 in array A and 1 in array B. It is made 0 in the newly constructed array D.

The array D will still be impossible but it now contains a number of optimized locations. These have been found with no comparison at all. The reason for not optimizing location y directly by replacing it with a 1 is that this location has not been checked to see if it is isolated. By leaving location y as 0 and continuing the synthesis procedure from this point, this check is made as a matter of course.

The array D may or may not contain 8's and ø's. The next step in the procedure after the generation of this array is to check for this property. If it does contain 8's and/or ø's it is an incompletely specified array and is placed in INCAR, the incompletely specified array list. If it does not contain these variables it continues on its way to have more locations optimized. The first comparison to be made will be with array C. Since array C is plausible and array D is not, array C should be the superior. If the choice of JTHLD is wrong this may not result. If JTHLD is too great the error will be corrected in subsequent operations. If JTHLD is too small, utilising the link-limit may not result in the optimum.

To illustrate the second use of the link-limit consider the situation illustrated by figure 3-21. The arrays in this figure are purely hypothetical, they represent no real network.

(000100100010)	Array A	KOMPAR	KLO
		(000100101010)	6
	Array C	(000100110010)	5
		(1XXXXXXXXXXX)	0
		(01XXXXXXXXXX)	1
		(001XXXXXXXXX)	2

Figure 3-21. Example Showing the Impossibility Test

It can be seen by comparing arrays A and C that y is location 8. Array B is constructed by first placing 1's in all locations which are 8's or ø's in array A. Then placing 1's in all locations specified as 0's in array C. Array B appears as below:

(111111100011) Array B.

This is the test array. All arrays generated from array A will have at least the 0's present in array B. Say that array B is judged implausible indicating that Array A is impossible. Thus all the arrays generated from array A are implausible. Array D is formed by combining arrays A and C in the manner previously described.

(000100100010) Array D

This array replaces array A in the synthesis procedure. By using the link-limit all the array generations comparisons and cost calculations that would have been required to optimize the five locations have been eliminated. This is an extremely significant savings especially apparent for large networks and small values of JTHLD.

Whether JTHLD is too great or too small depends on the maximum number of error-linked sources to a restored function in the optimum network (the one which would be derived without using the link-limit). If JTHLD-1 is greater than this maximum, the procedure should yield the optimum network. If JTHLD-1 is set less than this maximum the result may not be the optimum. Such a condition may be flagged if the result of the procedure has one or more restored functions with exactly JTHLD-1 error-linked sources. If this occurs JTHLD might have been set too low, and the synthesis should be tried again with a greater JTHLD.

For systems with feedback, under some conditions a poorly chosen JTHLD will not be so easily flagged. If the optimum design has no restorers in a feedback loop, and the JTHLD is less than the number of functions in the loop, the procedure may yield a design with several restorers in the loop. There may be no restored functions with exactly JTHLD-1 error-linked sources.

Although the link-limit tests have been included in the computer program to perform synthesis, they have not been thoroughly tested to measure their advantages. They will yield considerable reductions in the time required for synthesis, however, especially when restorers are fairly close together in the optimum network.

F. THE APPROXIMATIONS IN THE PROCEDURE

The Isolating Array Synthesis Procedure does not find the network which minimizes the True Cost of equation 4. The technique uses an approximation of this cost, so that the characteristic of isolation can be used to considerably reduce the number of calculations that must be performed in the optimization procedure.

Assume a set of functions are chosen from the network and called members of the set Q. The locations of the set Q are to be optimized, but for the moment let the locations of every member of the set Q be empty. Now let the locations of the functions not in the set Q take on an array of states with some locations filled and some empty.

The functions and restorers not in Q can be divided into two sets, E and I. A member of set E is error-linked to at least one of the members of set Q and a member of set I is isolated from every member of Q. Three disjoint sets have now been defined.

The approximation is based on the method of computing the reliability of redundant networks which is described in detail in Appendix A of the First Annual Report. This appendix shows that the reliability of a network, R, can be factored into two terms R_I and R_E such that:

$$R = R_I R_E.$$

R_I consists only of factors which contain the reliabilities of circuits in functions in the set I, outside the isolated region. R_E consists only of factors which contain the reliabilities of circuits in functions in the sets E and Q, inside the isolated region.

Now say the states of the locations within the set Q are to be optimized, with the locations not in set Q specified as some array. When a restorer is placed in a location, it takes on the function's outputs. The functions and restorers that were error-linked to the function alone are now error-linked to the function or its restorer or perhaps both, but no new functions are error-linked with the combination. Then, as restorers are added to the set Q, the sets I and E remain the same. No new terms are introduced into R_I, so it does not change, but R_E changes because of changes in the set Q.

The optimum array of the states of the locations in the set Q (given the array of the locations not in Q) is that which minimizes the True Cost. The functional costs of the sets Q, I, and E are independent and are represented by F_Q, F_I, and F_E respectively. The True Cost for this network is written:

$$\text{True Cost} = F_I + F_E + F_Q + (1-R_I R_E)\,K. \tag{6}$$

Since only the members of set Q are allowed to change, only the terms F_Q and R_E of the True Cost will vary. Say there are two different arrays of the locations in Q, Q' and Q'', whose True Costs are being compared. The difference between the cost of the two networks is:

$$\begin{aligned}\text{True Cost}(Q') - \text{True Cost}(Q'') &= F_Q(Q') - F_Q(Q'') \\ &+ \left[1-R_I R_E(Q')\right] K - \left[1-R_I R_E(Q'')\right] K \\ &= F_Q(Q') - F_Q(Q'') + R_I \left[R_E(Q'') - R_E(Q')\right] K \end{aligned} \tag{7}$$

For most situations, the value of R_I will be very close to 1 and will have very little effect on the decision between the arrays Q' and Q''. Then the approximate difference between the true costs of the two arrays is:

$$\text{True Cost (Q')} - \text{True Cost (Q'')} \approx F_Q(Q') - F_Q(Q'') + \left[R_E(Q'') - R_E(Q')\right] K. \quad (8)$$

The optimum array of the locations in set Q found using this equation is independent of the functions or restorers in the set I. The equation affirms that the optimum state of a set of locations does not depend on the functions or the state of the locations that are isolated from the functions in the set. This is a very important approximation and it is the crux of the Isolating Array Synthesis Procedure. Its use considerably reduces the number of calculations the designer will have to make for the synthesis of a large multiple-line network.

In the procedure, the only costs calculated are the costs of the isolated regions made up of the sets Q and E. This cost is called the region cost and is:

$$\rho(Q') = F_Q(A') + F_E + \left[1-R_E(Q')\right] K \quad (9)$$

The region cost is independent of the set I. The difference between region costs for the arrays Q' and Q'' results in equation 8.

In almost all cases the assumption that R_I is equal to 1 will not seriously effect the results of the synthesis procedure. If the result is different from the True Optimum, the cost of the resulting network will probably not be much greater than the minimum True Cost.

The approximation assumes that the optimization of a small isolated region, independent of the rest of the network, is consistent with the optimization of the whole network. The approximation optimizes a region, assuming the rest of the network is perfectly reliable. The difficulty is that the actual cost of failure of the isolated region is dependent on the reliabilities and location states outside the region. For example, consider the extreme case where the reliability of circuits in a function outside the isolated region is zero. These circuits cannot operate correctly, and the network is surely failed. A restorer added to minimize the cost of a region is really no help at all to the total network, since it has already failed. The restorer can only add to the functional cost of the network. The procedure does not take into consideration functions outside the isolated region, so the result of the procedure in this case may not be optimum.

This extreme example has turned up the approximation in the procedure, but this is not serious. A good portion of the network need not be considered when determining the optimum state of a location. This proves to be so valuable an attribute that it far outweighs the approximation brought to light in this section.

It does only lead to a slight approximation, because almost all networks that will be synthesized will have extremely high reliability specifications. When optimizing a location, using some small portion of the network, it is not very erroneous to assume that the rest of the network is working. Under this assumption, the procedure as described so far is perfectly valid.

When K, the cost of failure, is so high that the goal of the design is to maximize the reliability of the network, the procedure is valid without any approximation. No matter what the reliabilities of the functions outside the region which includes some function y the optimum state of location y is the one which minimizes the probability of failure of the region and therefore, minimizes the expected cost of failure of the region. Utilizing the optimum cannot decrease the reliability of the network, and it will increase it if the rest of the network is not sure to fail.

G THE COMPUTER PROGRAM

The synthesis procedure has been programmed for implementation on an IBM 7094. The program uses FORTRAN 2 and FAP coding. The FAP was included to increase the speed of the program and to reduce the amount of memory required in the synthesis of large networks.

The program has run for simple networks having up to 50 functions. The networks are simplified to shorten the time required for debugging runs. Final debugging will allow the synthesis of networks having up to 98 functions. This upper limit is easily expanded by changing a few limits in the program.

The program has not been reproduced for this report, but a flow diagram outlining the main steps of the program can be found in Appendix B.

IV. OTHER USES FOR THE SYNTHESIS PROCEDURE

If the discussions of this paper are couched in more general terms, it becomes apparent that the isolating array synthesis procedure is applicable to problems other than finding the optimum arrangement of restorers in a redundant network. In fact, the procedure is applicable to a whole class of problems, the characteristics of which are described in this section.

Consider the problem in which a large number of decisions are to be made. Each decision is binary in that there are two alternatives available, and one must be chosen. All decisions are alike in that the same alternatives are available for each. The problem is to choose values for each decision to optimize some parameter. For the redundancy application this parameter is True Cost and the decision to be made is whether or not to include a restorer at a location. The characteristic which makes this problem a difficult one is that no strict subset of decisions can be made independently of all other decisions.

Of course, finding the optimum placement of restorers in multiple line networks is one application of the synthesis procedure. To illustrate that the class of problems also amenable to solution by this means is not empty, a hypothetical quality control problem is here presented.

Let the diagram of figure 4-1 represent a manufacturing process. The nodes represent individual operations in the process, and the line segments indicate the sequential order of the operations. This process is manufacturing a product which cannot be repaired economically if it is found faulty at some point in manufacture. Perhaps it is a line for integrated electronic circuits. Thus, a faulty product is thrown away.

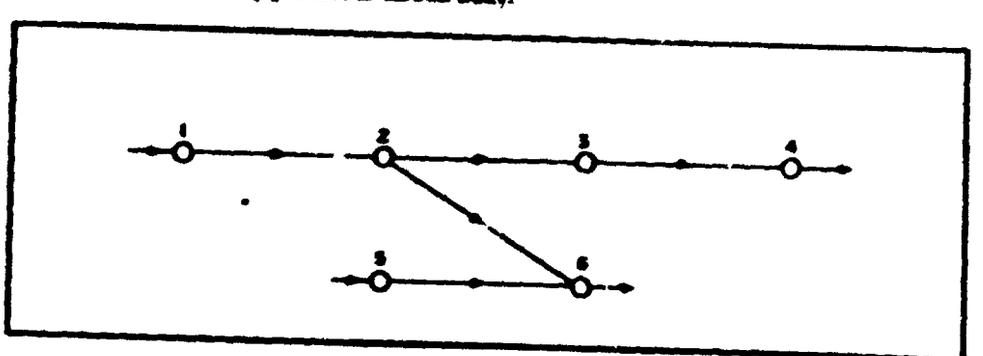


Figure 4-1. Flow Diagram for Quality Control Problem

After each operation it is possible to exhaustively test all the products that arrive at that point and throw away all that are faulty. This is called the quality control test. The cost of the test will depend on the complexity of the product so there will be a cost of test per product

associated with each operation. The cost of the unit thrown away depends on how far it has progressed through the manufacturing process. Thus, there is a cost associated with each operation which represents the cost of throwing away the product after that operation.

Each operation in the process has a constant probability of producing a faulty unit. The probabilities may differ for different operations. The number of products thrown away in the manufacturing process is assumed small relative to the number passing through it, so it is assumed that there is the same number of products passing through each operation. With these assumptions the expected cost per product of performing a quality control test after operation i is:

$$p_i C_i + Q_i$$

Where p_i = the probability of the product being faulty at the ith operation.

C_i = the cost of throwing away the product after the ith operation. Q_i = the cost of performing the quality control test on the product after the ith operation.

If quality control tests are performed after every operation in the set M, the total cost due to faulty manufacture is:

$$\sum_{i \in M} p_i C_i + Q_i$$

The problem is to minimize this cost by optimally making the decision at each operation whether or not to make a quality control test. The decisions cannot be made independently because the value of p_i at any operation depends on where the fault eliminating quality control tests have been performed previous to the operation. Thus, the decision cannot be made at a particular operation without considering what other decisions have been made. In fact, no strict subset of decisions can be made independently of all other decisions.

The solution to this problem can be found with the synthesis procedure. There are other examples one might postulate that will illustrate the class of problems to which the procedure is applicable, but this one and the redundancy example should be sufficient to at least show that the class is not empty.

A. GENERAL CHARACTERISTICS OF THE CLASS

With the help of the two examples, the general characteristics of applicable problems will now be pointed out.

1. Form

The problem must be representable by a network diagram such as shown in figure 4-2. The nodes and line segments should describe the physical aspects of the situation under study and the relationships between them.

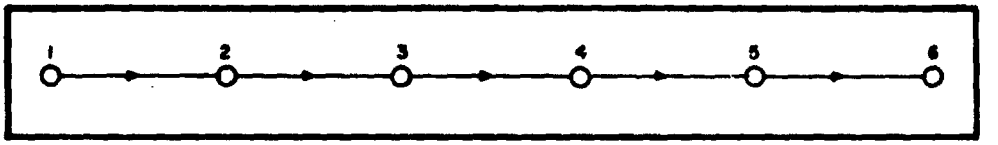


Figure 4-2. Simple Network Diagram

2. Decisions

At each node a binary decision must be made. The decision determines the inclusion or exclusion of some object or operation. The decision does not modify the network diagram in any way, but it affects some parameter of the network. It is impossible to determine the effect of some strict subset of decisions without knowledge of some decisions not in the subset. The decision for the redundancy example was whether or not to include a restorer after a function; and for the quality control example, it was whether or not to provide a quality control test after an operation.

Given the state of all but one of the decisions, it should be relatively simple to make the remaining decision. This restriction is included to keep the problem within bounds. The synthesis procedure requires that many determinations of the optimum decision for one node be made with knowledge of the state of the other nodes. If one of these determinations is difficult to perform, the procedure will take too long to implement, hence it will be impractical.

3. Parameter to be Optimized

There is some calculable parameter of the network which is affected by the decisions made at the nodes. The goal of the procedure is to optimize this parameter by determining the best set of decisions at the nodes. The parameter for the first example was True Cost, and for the second it was cost.

4. Isolation Occasioned by an Affirmative Decision

The characteristic of isolation provided by a restorer (an affirmative decision) has been described in Section VII. B. 2 of the report. To illustrate the characteristic for the quality control example consider the simple network diagram of figure 4-2.

Assume the network diagram of figure 4-2 represents a manufacturing process. An affirmative decision at node 2 represents the inclusion of a quality control test after operation 2. This test removes all faulty products from the process at this point, hence the probability that a product is faulty at nodes 3, 4, 5 or 6 is independent of the decision made before node 2. The expected cost due to faulty manufacture can now be calculated independently for the networks before and after node 2, and their sum will be the total expected cost. The affirmative decision has resulted in isolation.

5. Isolation Not Occasioned by a Negative Decision

This characteristic prevents the problem from becoming a trivial one. If both negative and affirmative decisions resulted in the isolation of decisions, each decision would be completely isolated from every other decision. This would make it possible to make each decision independently of all others. Of course the synthesis procedure is not needed for problems in which all decisions can be made independently.

The final two characteristics deal with negative and affirmative decisions. Of course, by the reversing the definitions of negative and affirmative, the characteristics can just as well read that negative decisions result in isolation and affirmative decisions do not. If a problem posesses these characteristics it is amenable to solution by the Isolating Array Synthesis Procedure.

V. CONCLUSION

The Isolating Array Synthesis Procedure has been developed to the point where it is now available for evaluation and use. The program to implement the procedure on a digital computer has been run successfully for several example networks. The first large scale synthesis task is being readied.

The time required for synthesis depends primarily on the number of functions in the network, the interconnection pattern between the functions and the value assigned to JTHLD, the maximum number of error-linked functions providing the inputs to a restorer. At this time the number of networks synthesized with the procedure is insufficient to estimate the time required.

Several projects still remain to be performed. The program for the synthesis procedure provides a tool for its own evaluation and the discovery of insights concerning the placement of resoters in redundant networks. A carefully designed test schedule should reap considerable benefits in the understanding of redundant networks.

The discovery of a procedure to concurrently optimize the order of redundancy of the functions of the network and the placement of restorers within it has not yet been accomplished. However, some initial probes into this area have indicated that this additional task can be done with some modifications of the present procedure. The accomplishment of this task will allow the different functions in the network to assume different orders of redundancy in an optimum manner. Particularly important is the ability to find the optimum network in which only order three redundant and non-redundant functions are allowed. A procedure with this ability appears to have the most immediate application. Some future study should be devoted to this problem.

This study has developed a tool of significant value to engineers charged with the design of redundant-multiple-line networks. The procedure finds the optimum placement of restorers in networks of arbitrary topology. This procedure is a significant contribution to the state-of-the-art because it performs an optimization never before achieved except by exhaustive search. The techniques described herein may also find applications in the solution of other problems which are concerned with making a large set of binary decisions in an optimum manner.

APPENDICES

Appendix A

EXAMPLE DESIGN

To illustrate the performance of the synthesis procedure, this example design is presented in figure A-1. It shows both the array generation procedure and the array comparison procedure. The arrays that are generated by comparisons and contain optimized locations are underlined. The comparisons that have been made are noted for the first branch of the tree. The symbol c indicates a comparison of true costs. An equation of the form: (9) - [(8) c (3)] c (7) indicates the arrays which take part in the comparison and the order in which the comparisons are performed. For instance, this equation indicates that array 9 is formed by first comparing arrays 3 and 8, and then comparing the result of this comparison with array 7.

For the example JTHLD is set equal to 3, so regions with four or more error-linked functions are not considered in the synthesis. The possibility test is illustrated but the plausibility test cannot be shown in such a diagram.

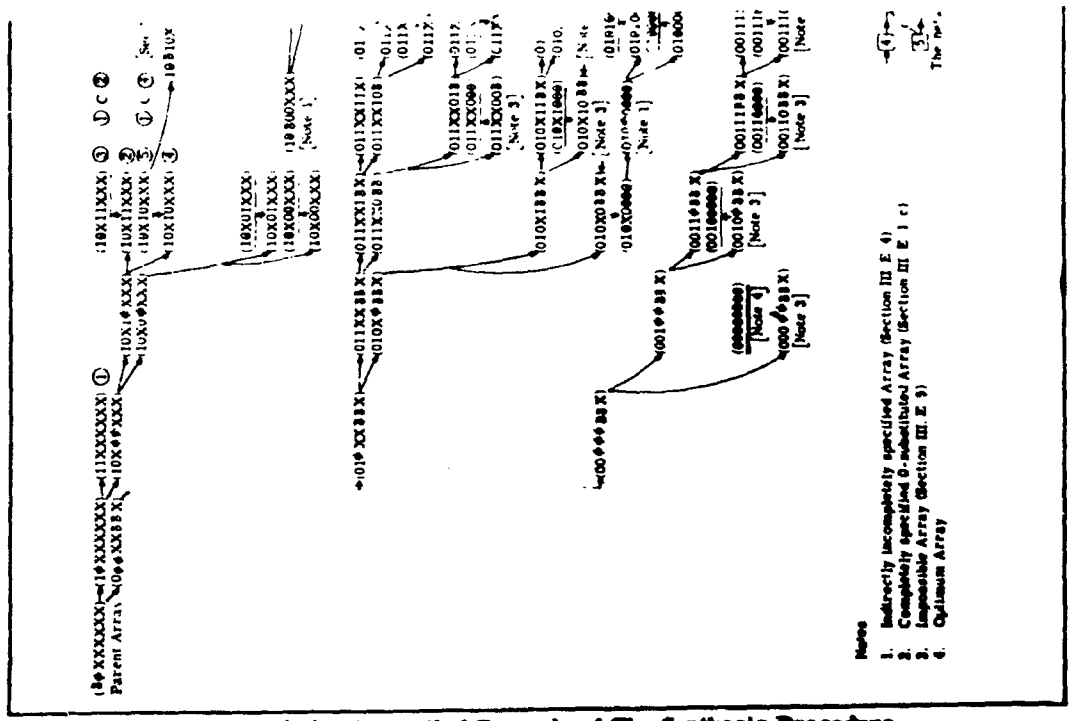
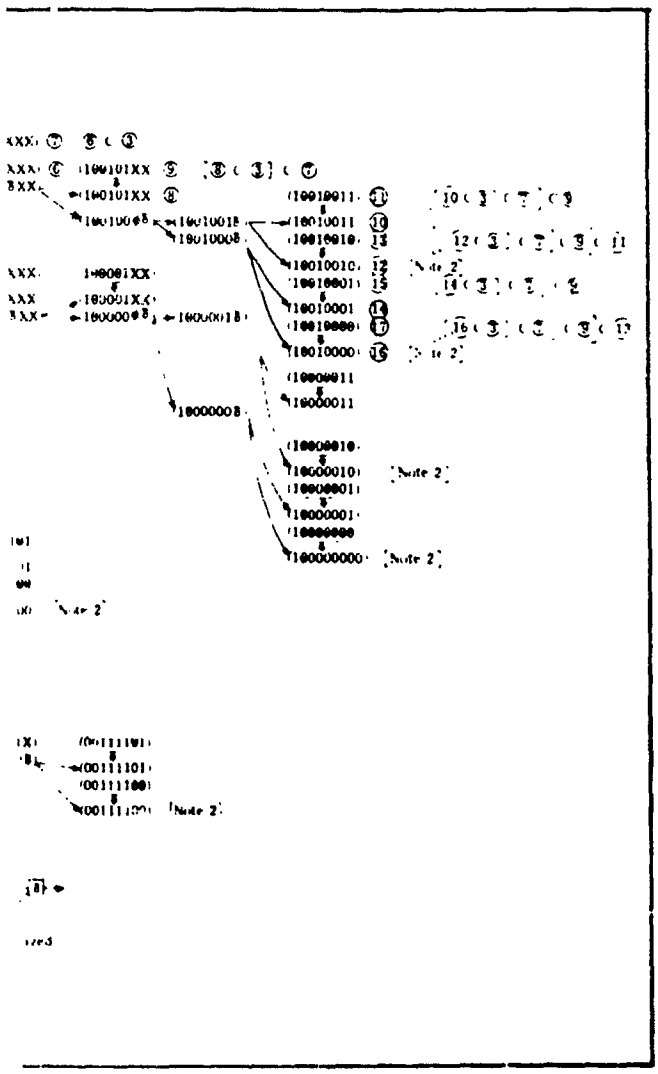


Figure A-1. A Detailed Example of The Synthesis Procedure



Appendix B

A DESCRIPTION OF THE PROGRAM FOR THE COMPUTER IMPLEMENTATION OF THE SYNTHESIS PROCEDURE

I. GENERAL DISCUSSION

The following serve as inputs to the program:

1. Connections of the network
2. Threshold for the link-limit
3. Order of redundancy
4. Number of functions
5. For each function and restorer, the cost of:
   a. Implementation
   b. Power
   c. Weight
6. Cost of failure (K)
7. Reliabilities of each function and restorer
8. Minimum number of lines for successful operation
9. The number and list of outputs of the network.

The output of the program is an array showing the optimum placement of restorers in the network.

Figure B-1a shows a six-element network whose optimization tree is manually developed as an example in the discussion of the program which follows. A general flow diagram of this program is illustrated in figure B-2.

Input information to the program is entered at block A. Reliabilities and costs of various arrays are calculated in the block at the bottom which is designated as K. The remainder of the program serves to develop the tree according to the rules of the synthesis procedure and to route the various generated arrays to their proper places.

A. THE CONNECTION TABLE (ICONT)

Let us now examine how the configuration of the network to be optimized is converted into computer input information. For the network in figure B-1a a connection table of the type shown in figure B-1(b) may be constructed. In essence, this table is a matrix where the columns represent the outputs of the designated functions and the rows the inputs to the designated functions. A "1" represents a connection. Thusly, it may be seen from figure B-1a and B-1b, that a "1" in (2, 1) means that the output of function 1 is the input of function 2.

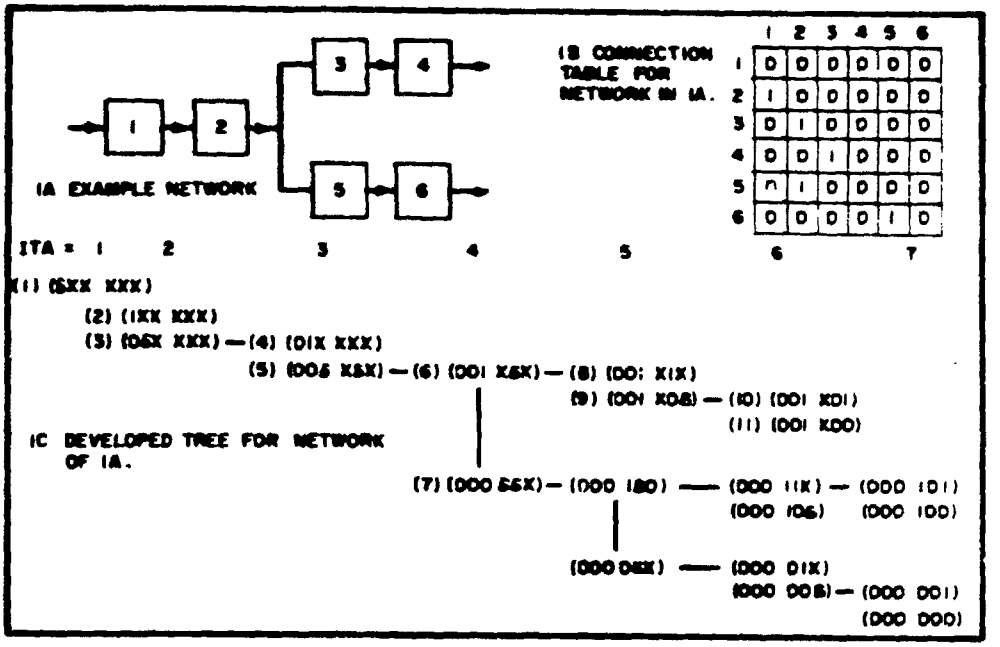


Figure B-1. Example for the Description of the Computer Program

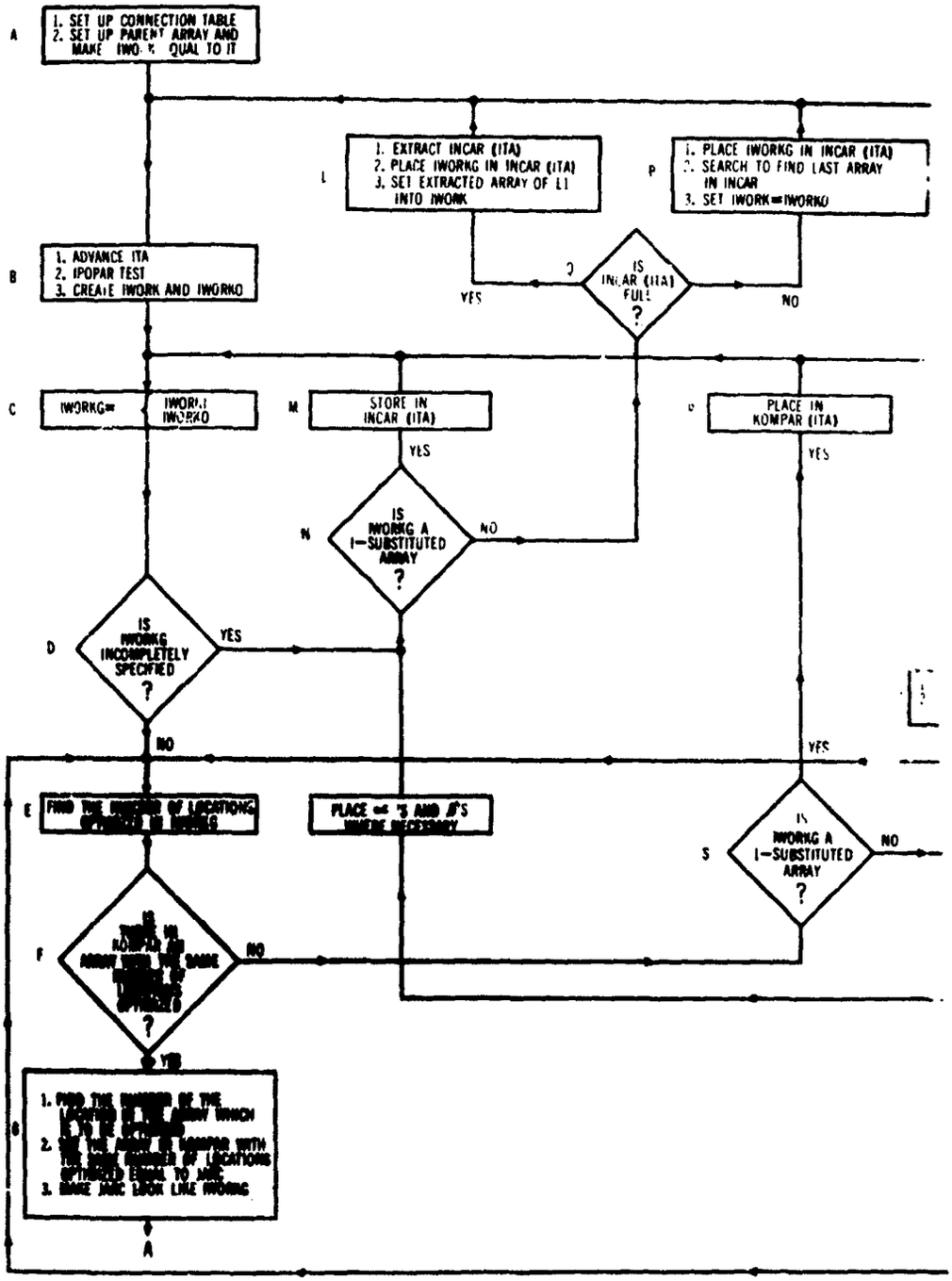
Likewise, the 1's in (3, 2) and (5, 2) signify that the output of 2 serves as input to functions 3 and 5. Similar information is in this way entered into, and stored as, a matrix in the computer memory for all the remaining functions of the network. All other locations in the connection table, those not equal to 1's, are set to 0.

B. THE PARENT ARRAY

The parent array is formed from the connection table by placing a 6 in location 1 of the array, a ϕ in every location where a "1" appears in the first row of the connection table, and an X in all remaining locations.

C. ITA

For reasons which will later become more evident, a measure of distance from the parent array is made along the tree (see figure B-1) as it develops. This distance measurement is in the form of ascending integers called ITA. Thus, the parent array is at ITA 1. the next group of arrays at ITA = 2, etc.


A
1. SET UP CONNECTION TABLE
2. SET UP PARENT ARRAY AND MAKE IWO.. QUAL TO IT
L
1. EXTRACT INCAR (ITA)
2. PLACE IWORKG IN INCAR (ITA)
3. SET EXTRACTED ARRAY OF L1 INTO IWORK
P
1. PLACE IWORKG IN INCAR (ITA)
2. SEARCH TO FIND LAST ARRAY IN INCAR
3. SET IWORK=IWORKO
B
1. ADVANCE ITA
2. IPOPAR TEST
3. CREATE IWORK AND IWORKO
Q
IS INCAR (ITA) FULL ?
YES
NO
C
IWORKG= IWORK IWORKO
M
STORE IN INCAR (ITA)
YES
N
IS IWORKG A I-SUBSTITUTED ARRAY ?
NO
P
PLACE IN KOMPAR (ITA)
YES
D
IS IWORKG INCOMPLETELY SPECIFIED ?
YES
NO
YES
E
FIND THE NUMBER OF LOCATIONS
PLACE ∝ 'S AND β'S
S
IS IWORKG A I-SUBSTITUTED ARRAY ?
NO
F
NO
YES
G
A

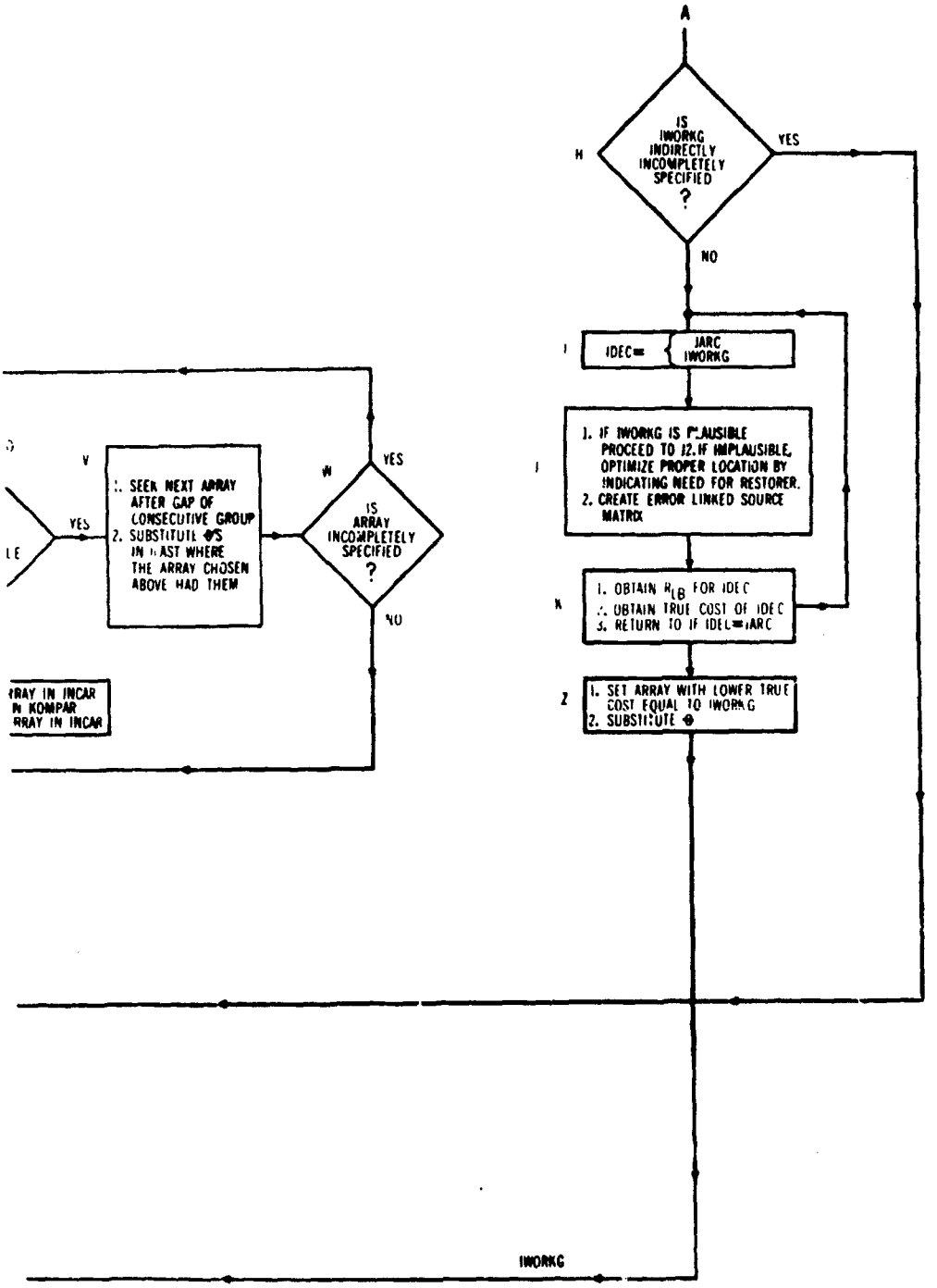


Figure B-2. General Flow Diagram

D. KOMPAR

Completely specified arrays, when stored, are kept in a block in memory designated as KOMPAR. This block is a two dimensional matrix in which completely specified arrays are stored in locations corresponding to their ITA values. The form which this matrix takes is shown in figure B-3. Thus, for example, the array in KOMPAR (5) is (001 X1X).

ITA	ARRAY
1	(000 000)
2	(1XX XXX)
3	(01X XXX)
4	(000 000)
5	(001 X1X)
6	(001 X01)

Figure B-3. An Example of Arrays Stored in KOMPAR

E. KLO

A count of the number of locations optimized is made for each array in KOMPAR and this number is stored in a list called KLO. The ITA value of the array with the given number of locations optimized is also included in the KLO list. The form of this list is shown in figure B-4.

LOCATIONS OPTIMIZED	ITA
0	2
1	3
2	5
3	6

Figure B-4. The KLO list

If we desire to find an array in KOMPAR with no locations optimized we need only search through the KLO list to find that such an array exists at ITA = 2. Now looking through KOMPAR for the array at ITA = 2, we can extract (1XX XXX), which indeed, has zero locations optimized.

F. INCAR

Incompletely specified arrays are stored in a similar manner as are ones which are completely specified. The block in memory in which the incompletely specified arrays are stored is called INCAR, and takes the form of figure B-5. Arrays are placed in INCAR as shown. Whenever they are extracted, the location is filled with zeros.

ITA	ARRAY
1	(000 000)
2	(0$X XXX)
3	(000 000)
4	(001 X$X)
"	"
"	"
"	"

Figure B-5. INCAR With Arrays Stored at ITA = 2 and ITA = 4

G. SYMBOL CODING

The symbols 1, 0, $, ¢, X, θ_1, and θ_0 are represented in the program by the numbers 0 thru 6 as shown in figure B-6.

SYMBOL	CODE
0	0
1	1
θ_1	2
θ_0	3
X	4
$	5
¢	6

Figure B-6. The Coding Symbols of the Program

Thus the array (10X $ 0¢$) is converted to the representative numerically-coded array (1045065). Likewise (θ_1 θ_0 XX XXX) is represented by (2344 444).

H. PACKING AND UNPACKING

In FORTRAN it would be necessary to utilize one storage location for each code number in an array. Therefore, the array (2344 444) would require seven words, the first one "2", the next "3", etc. By using FAP subroutines, arrays such as the above can be neatly packed into single words, effecting a twelve times reduction in array storage requirements. In the above example the array (2, 3, 4, 4, 4, 4, 4) would be packed as (234444400000).*

Whenever the number of symbols to be packed exceeds 12, more words are added. For example, when the program is packing a 100 symbol array, nine words must be assigned to the array. The eight extra locations needed to fill the ninth word, a partially completed word, are filled with zeros.

When an array is to be used in the FORTRAN program it must first be unpacked so that each symbol is designated by its own word. A special unpacking subroutine is used to accomplish this.

I. IPOPAR TEST

Assigning 1's and 0's to δ's and β's in an incompletely specified array must be done in proper order, so that those locations previously optimized must be varied last. (See Section III. E. 3). A record of these previously optimized locations is kept in an array in memory which is called IPOPAR. The IPOPAR test must then necessarily precede the assigning of 1's and 0's to incompletely specified locations in IWORK.

J. DEVELOPMENT OF THE TREE

We are now ready to see how the tree illustrated in figure B-1c is developed by the program of figure B-2, as the program proceeds thru its various paths. An enumerated description of these paths now follows.

PATH 1

Block	Step	
A	1.	Input information describing the network enters the computer in the form of a connection table.
	2.	The ITA is set to 1.
	3.	From the connection table the parent array is formed (figure B-1c, (1)) and set into IWORK. The flow now enters B.

* The maximum number of symbols which may be stored in the IBM 7094 memory word is 12. Each number is in the octal system.

PATH 1 (Continued)

Block	Step	
B	1.	The ITA number is advanced to 2.
	2.	The IPOPAR test is made to determine which location is to be varied.
	3.	First a 1 is substituted for that location and the array is termed a "1-substituted array". It is designed in the program as IWORK1. Likewise an array termed the "0-substituted array" is generated by substituting a "0" in the proper location and adding the appropriate δ's and ∅'s. The array is named IWORK0. IWORK1 and IWORK0 are represented in figure B-1c as arrays (2) and (3), respectively.
C	1.	On the first pass into block C, IWORKG is created by being set equal to IWORK1. On the second pass IWORKG will become IWORK0.
D	1.	A test of the specification of IWORKG is made here. Since IWORKG is array (2) in figure B-1c and is not incompletely specified (i.e., it is completely specified) we enter E instead of H.
E	1.	The number of locations optimized in IWORKG (number of θ's) is determined. In this case the number is 0.
F	1.	In order to find a suitable array to compare with IWORKG, a search is made thru the KLO list for an array in KOMPAR with the same number of locations optimized i.e θ's. Since this is the first path, no previous arrays exist to enable a comparison, and the flow exits F into S.
S	1.	The array is a 1-substituted array.
R	1.	The array is placed into KOMPAR (2), where "2" is the present ITA number. It appears in figure B-3 at ITA of 2 (arrays with higher ITA's do not exist at this point of the program). Furthermore, a note is made in the KLO list (figure B-4) that in KOMPAR of ITA-2, an array exists with 0 locations optimized.

Path 1 now ends with a second entry into C.

PATH 2

Block	Step	
C	1.	Since this is the second entry into C, IWORKG now becomes IWORK0, which is (3) in figure B-1c.
D	1.	IWORKG is incompletely specified and the flow exits into H.
N	1.	The array is 0-substituted and we proceed to Q.
Q	1.	INCAR of ITA=2 is empty since this is the first incompletely specified array encountered at this ITA number.
P	1.	IWORKG is placed in INCAR (2) as shown in figure B-7.
	2.	A search through INCAR for the last array in it, is made. In this case, the last array is the array just placed into INCAR in 1, above. This array is now entered into the array called IWORK, as the program completes PATH 2 on its entry into B.

ITA	ARRAY
1	(000 000)
2	(0?X XXX)
3	(000 000)
4	(000 000)
"	"
"	"
"	"

Figure B-7. INCAR With an Array at ITA = 2

PATH 3

Block	Step	
B	1.	ITA is increased from 2 to 3.
	2.	IWORK, which is array (3) in figure B-1c, undergoes the IPOPAR test to find which location must next be varied.
	3.	IWORK1 and IWORK0 are created by the substitution of a 1 and a 0 respectively in the location found in 2 above. In figure B-1c these arrays are (4) and (5) respectively.

PATH 3 (Continued)

Block	Step	
C	1.	This being the first pass into C for the new IWORK1 and IWORK0, IWORKG is set equal to IWORK1.
D	1.	IWORK1 is (4) in figure B-1c and is not incompletely specified. The flow proceeds to E.
E	1.	The number of locations optimized in IWORKG is 0.
F	1.	There is an array in KOMPAR with no locations optimized. The KLO list reveals it is at ITA-2. In KOMPAR (2) the array (1XX XXX) is listed and is array (2) in figure B-1c.
G	1.	In the comparison of the KOMPAR array and IWORKG, the location to be optimized is the one which contains a "0" in one of the arrays and a "1" in the other array. This location is found here.
	2.	The array found in KOMPAR (2) is extracted by setting it into an array called JARC.
	3.	JARC is made to look like IWORKG by changing all those locations which are "X" in JARC to those symbols occupying the same locations in IWORKG. Thus, in this path the second location in array (2), figure B-1c, assumes the value of the second location of (4), figure B-1c. JARC and IWORKG are now ready for comparison.
H	1.	IWORKG is not indirectly incompletely specified and the flow enters I.
I	1.	In order to be compared for cost, IWORKG and JARC are made equal to IDEC one at a time i.e. the cost is first found for IDEC = IWORKG and then for IDEC = JARC. This first time, then, IDEC = IWORKG.
J	1.	The plausibility is performed for IWORKG. If IWORKG is implausible the location to be optimized is set to 0, and the flow skips to E1 in path 4.
	2.	If plausible, the error-linked source matrix* is created for IDEC = IWORKG.

* The error-linked source matrix is used in the reliability analysis as described in the First Annual Report.

PATH 3 (Continued)

Block	Step	
K	1.	Lower-bound reliability is computed for IWORKG.
	2.	True cost is obtained for IDEC and stored.
	3.	Since IDEC = IWORKG, the flow returns to I.
I	1.	IDEC = JARC
J	1.	The error-linked source matrix for IDEC = JARC is created.
K	1.	Lower-bound reliability is obtained.
	2.	True cost is computed for IDEC and stored.
Z	1.	The array with the lower true cost is chosen and set into IWORKG as the flow returns to E and the path is completed.
	2.	If the 1-substituted array is chosen as most optimum, then θ_1 is substituted in the location to be optimized. If the 0-substituted array is optimum a θ_0 is placed in the location to be optimized.

PATH 4

Block	Step	
E	1.	The number of locations optimized in IWORKG is one.
F	1.	In KOMPAR there is no array at this moment with one location optimized.
S	1.	Array (4) in figure B-1c, is a 1-substituted array.
R	1.	The array is placed into KOMPAR of ITA = 3. The program returns to C to complete path 4. The array appears in KOMPAR as shown in figure B-3 at ITA = 3 and its number of locations optimized (one) appears in the KLO list with ITA = 3.

PATH 5

Block	Step	
C	1.	As the flow enters C for the second time IWORKG becomes IWORK0. This is array (5) in figure B-1c.
D	1.	IWORKG is incompletely specified.

PATH 5 (Continued)

Block	Step	
N	1.	IWORKG is NOT a 1-substituted array.
Q	1.	INCAR of ITA = 3 is not full since no array has previously been placed into this position.
P	1.	IWORKG is placed into INCAR (3).
	2.	The last array is sought in INCAR.
	3.	This last array, which is the one inserted in P1, is pulled out of INCAR and set into IWORK. Figure B-8 shows INCAR (3) as array (5) of figure B-1c.

ITA	ARRAY
1	((000 000)
2	(000 000)
3	((006 X&X)
4	((000 000)
"	"
"	"
"	"

Figure B-8. INCAR With An Array at ITA = 3

PATH 6

Block	Step	
B	1.	ITA is advanced to 3.
	2.	The IPOPAR test indicates which location is to be varied.
	3.	A "1" and "0" are substituted for this location to create IWORK1 and IWORK0. These represent arrays ((6) and ((7) in figure B-1c, respectively.
C	1.	IWORKG becomes IWORK1.
D	1.	IWORKG is incompletely specified.
N	1.	IWORKG is a 1-substituted array.
M	1.	IWORKG is stored in INCAR ((4). See figure B-9.

ITA	ARRAY
1	(000 000)
2	(000 000)
3	(000 000)
4	(001 XXX) . . .
"	"
"	"
"	"

Figure B-9. INCAR With An Array (6) at ITA = 4

PATH 7

Block	Step	
C	1.	IWORKG becomes IWORK0.
D	1.	IWORKG is incompletely specified.
N	1.	IWORKG is a 0-substituted array.
Q	1.	INCAR ((4) is full. It contains array (6) figure B-1c.
L	1.	INCAR ((4) is extracted.
	2.	IWORKG is placed in INCAR ((4) and appears in figure B-10.
	3.	The extracted array of L1 is placed into IWORK.

ITA	ARRAY
1	((000 000)
2	((000 000)
3	((000 000)
4	((000 XXX)
"	"
"	"
"	"

Figure B-10. INCAR With Array (7) at ITA = 4

PATH 8

Block	Step	
B	1.	ITA is advanced to 5.
	2.	The IPOPAR test determines which location is to be varied.
	3.	A "1" and "0" are substituted in this location to form IWORK1 and IWORK0.
C	1.	IWORKG becomes IWORK1 (array (8) , figure B-1c).
D	1.	IWORKG in not incompletely specified.
E	1.	The number of locations optimized is 0.
F	1.	There is an array in KOMPAR with an equal number of locations optimized. It is at ITA = 1.
G	1.	The location is 1.
	2.	JARC becomes array (1) figure B-1c.
	3.	JARC is made to look like IWORKG by placing a "0" a "1", and another "1", in locations 2, 3, and 5 respectively.
H	1.	IWORKG is not indirectly incompletely specified. The plausibility
J, K, Z		check is made. If implausible, the optimized array, containing a 0 in location 1, is set equal to IWORKG. The array is (001 X1X).

PATH 9

Block	Step	
E	1.	The number of locations optimized is 1.
F	1.	There is an array in KOMPAR with an equal number of locations optimized. It is at ITA = 3.
G	1.	The number of the location to be optimized is 2.
	2.	JARC becomes (01X XXX).
	3.	JARC is changed to (011 X1X).
H	1.	No
J, K, Z		The plausibility check is made. If implausible, IWORKG and JARC are compared and the least costly becomes the new IWORKG, with 0's in locations 1 and 2. The new IWORKG is, then, (001 X1X).

PATH 10

Block	Step	
E	1.	The number of locations optimized is 2.
F	1.	There is no array in KOMPAR with an equal number of locations optimized.
S	1.	The array is a 1-substituted array.
R	1.	IWORKG is placed in KOMPAR (5) and appears so in figure B-3.

PATHS 11 and 12

By similar method array 10 figure B-1c is generated and compared to array (1) to give an array with one location optimized. This partially optimized array is then compared with the array in KOMPAR (3) and that resulting array with the one in KOMPAR (5) to yield in KOMPAR (6) an array with 3 locations optimized (see figure B-3).

Array (11) is likewise developed until a comparison can be made with the partially optimized array of KOMPAR (6) to yield an array with four 0's. This array is the output of Z and is set to IWORKG.

PATH 13

Block	Step	
E	1.	The number of locations optimized in IWORKG is four.
F	1.	There is no array in KOMPAR with an equal number of locations optimized.
S	1.	IWORKG is not a 1-substituted array.
U	1.	A search thru INCAR reveals that the last array, which has never been extracted is at ITA = 4. This is array (7) in figure B-1c. The array is now extracted.
	2.	IWORKG is therefore placed in KOMPAR (4).
T	1.	A check is made to determine whether the last array, extracted in U1, is possible. If the array is possible, it is placed into IWORKG and the chain is generated.

PATH 13 (Continued)

Block	Step	
	T	If the array is not possible, the flow contines to V, where the array described in Section III. E. 5 is constructed. In W, the flow is routed to either C or E, depending on whether the array is incompletely specified or not.

In a similar manner the remainder of the tree is developed until eventually a search through INCAR for the last array reveals that there are no more arrays stored in INCAR. At this point the existing IWORKG will have all its locations optimized (either θ_1 or θ_0). This optimum array is then placed into IOPTAR and represents the optimized network and the desired final result.

The Indirectly Incompletely Specified Case

Section III. E. 4 discusses the occurrence of the Indirectly Incompletely Specified Case. In the program this case is determined when IWORKG is tested at H. A "yes" decision sends the flow to 0, where ∮'s and ∅'s are placed in the necessary locations. Upon leaving 0, the flow continues to H and the remainder of the program as already described.

K. REFERENCES

1. Jensen, P. A., W. C. Mann, and M. R. Cosgrove, "The Synthesis of Redundant Multiple-Line Networks", First Annual Report Contract NONR 3842(00), May 1, 1963 (AD No. 410573).
2. Mann, W. C., "Systematically Introduced Redundancy in Logical Systems", 1961 IRE International Conv. Rec. 9, Pt. 2, 241-263 (March, 1961).
3. Jensen, P. A., "Four Redundant Configurations", Westinghouse Electric Corp., Electronics Division, Advanced Development, December 1, 1961, (Report No. EE-2600).
4. McReynolds, J., "Evaluation of the Majority Principle as a Technique for Improving Digital System Reliability", Hycon Eastern, Inc., July 8, 1958, (HEI Publication No. M-577) First Annual Report for Contract NONR-2133(00).